INCREMENTAL COMPILATION AND ITS IMPLEMENTATION IN THE PECAN PROGRAMMING ENVIRONMENT GENERATOR

JAMES POPPLE

A thesis submitted in partial fulfilment of the
requirements for the degree of Bachelor of Arts (Honours)
at the Australian National University.

November 1987

Abstract

The methodology and developmental history of incremental compilation is discussed. The implementation of incremental compilation in the PECAN programming environment generator is discussed in detail. The PECAN environment generated for Pascal has been modified to support procedure-by-procedure compilation, and complete (traditional) compilation. The time efficiency of these compilation methods is compared with that of incremental compilation.

Declaration

Except where otherwise indicated

this thesis is my own work.

James Popple

November 1987

Acknowledgements

Many thanks to my supervisor, Dr Chris Johnson, for much valuable advice and assistance, and for helping me plough through C code which (like [Ghezzi 80]) was never actually meant to be read; to Dr Brian Molinari for technical information; to Dr Malcolm Newey, (soon to be Dr) John Ophel and Sandra Dutton for help with references; to Harriet Michell for help with the *Scribe* document preparation system; to Gerry Blanch and Roger Poulier for help preparing diagrams with *MacDraw*; to Inta Skriveris for help with laser-printing; to Tim Findlow who worked with PECAN during the year; and to Norval Hope for fruitful discussion and mind expansion.

To my parents.

Me literulas stulti docuere parentes.

Marcus Valerius Martialis
Epigrams, book ix, epig. 74.

Table of Contents

List of Figures

Chapter 1
Introduction

Incremental compilers are designed so that only part of a program under development need be recompiled after a change has been made to its source code. This can be effected in one of two ways:

- by choosing a structure of the language and recompiling that whole structure whenever part of the structure is edited; or
- by determining the smallest amount of recompilation required after each individual editing change and recompiling only that section of the source code.

Using the first method (generally) involves unnecessary recompilation, but determining what source code to recompile is trivial. The second method performs no unnecessary recompilation, but requires extra computation to determine what source code to recompile.

The aim of this thesis project is to compare the relative efficiencies of these two approaches. To this end, an existing system (the PECAN programming environment generator) has been modified so that it allows compilation to be performed using either of the two methods of incremental compilation. Several example programs were chosen and edited so that comparisons could be made.

Chapter 2 discusses these two approaches in detail, and examines the difficulties caused by a programming language's ability to use names. Factors which affected the development of incremental compilers, and their relationship to programming environments and syntax-directed editors are discussed.

Chapter 3 gives examples of a number of incremental systems, and discusses the role of attribute grammars in generating programming environments.

Chapter 4 gives a description of the PECAN programming environment generator.

Chapter 5 gives a detailed description of the implementation of incremental compilation within the PECAN system.

PECAN takes the second of the two approaches mentioned above; it determines the smallest amount of compilation necessary after each change to the source. Chapter 6 describes how the PECAN environment for Pascal has been modified to allow procedure-by-procedure compilation and complete compilation, in addition to its incremental compilation. A benchmark was chosen for comparing these methods, and the results of a number of tests are included.

Conclusions are drawn in Chapter 7.

Part of the project involved the implementation of a new window for PECAN which provides a view of the internal data structure used by PECAN's compilation module. That view is described in Appendix A. Listings of the files that provide the view are included.

Details of the modifications made to PECAN's compilation module, with program listings, are given in Appendix B.

Appendix C lists the programs used in the tests described in Chapter 6.

Appendix D gives a detailed description of Earley's parsing algorithm (the algorithm used by PECAN).

Chapter 2
Incremental Compilation

2.1. Definition of Incremental Compilation

The development of a program can usually be characterized by an extended sequence of repeatedly editing and compiling source code. The programmer will often recompile a program after having made only a small change to the source code. If there is a large amount of source code, and the changes made are relatively minor, the compiler will be wasting much time and effort compiling source code which has not been changed since the last time that the program was compiled.

It is desirable that the programmer should have the convenience of a recompiled version of the program, ready to execute, as soon as possible after a change is made to the source code. This is particularly true when the program is being debugged and the programmer wants to monitor the effect upon the program's behaviour of a small modification.

A compiler is *incremental* if it provides the programmer with a recompiled version of the program "by expending an amount of effort which is proportional to the size of the change made by the programmer."[1]

2.2. Deciding What to Recompile

2.2.1. The Recompilable Unit

Ideally an incremental compiler will recompile as little of the source code as possible after each modification. In this thesis, the term *recompilable unit* will be used to describe that structure in a programming language which is recompiled by an incremental compiler when a change is made.[2]

---

[1] *per* Earley and Caizergues in [Earley 72].

[2] The term *minimal separately compilable unit* is used in [Reiss 84a], and the term *smallest compilation unit* is used in [Fritzson 83a].

Consider the following hypothetical language: a program is composed (*inter alia*) of statements; statements may be composed (*inter alia*) of expressions; and expressions may be composed (*inter alia*) of integers, which are sequences of digits. If the language is defined so that no change to a statement can affect the meaning of any part of the program outside that statement, then the statement is chosen as the recompilable unit.

However, if the programmer changes the value of an integer by altering a single digit, it may be that the code produced by recompiling the enclosing statement differs from the corresponding previously-compiled code only in the manner in which it represents that integer. Even though the compiler is incremental, it has performed unnecessary recompilation; it could have achieved the desired effect merely by replacing the code representing the original integer with code representing the modified integer. Alternatively, altering a single digit may radically change the code which will be produced for the enclosing expression, and possibly the enclosing statement.

For example, assume that the following is a valid statement in this hypothetical language

```
IF X < 10 THEN
   GOTO Label1
ELSE
   GOTO Label2
```

If the integer constant is changed from a 10 to a 9, the object code generated for the entire (modified) statement will differ from the previously-compiled code only in its representation of the integer 9. However, if the variable X is changed to the integer constant 9, the object code generated to evaluate the new boolean expression ($9 < 10$) will be quite different from the code generated to evaluate the old boolean expression ($X < 10$); no code will be required to look up the value of X. Furthermore, if the compiler performs simple code optimization then the object code for the entire statement can be replaced by object code to represent the statement

```
GOTO Label1
```

because the new boolean expression ($9 < 10$) is tautologous.

2.2.2. Choosing the Smallest Recompilable Unit

Incremental compilers can be usefully divided into two classes based upon their approach to the problem of deciding what to recompile after each change. Some choose a syntactic unit of the language (independent of any particular program) as the recompilable unit. This recompilable unit is recompiled whenever a change is

made within that unit. Others attempt to determine the *smallest recompilable unit* (specific to the change being made) in order to be able to recompile as little as possible. These two approaches will be referred to as α-type and β-type respectively.

α-type incremental compilers will generally perform unnecessary recompilation after each change.[3] β-type incremental compilers will recompile only what is necessary, but incur considerable overheads in time and (usually) space in order to determine the smallest recompilable unit. The Magpie system (see §3.3.5) is an example of an α-type system. PECAN (see Chapter 4) is an example of a β-type system.

Balancing the costs of these two approaches is *the* fundamental question in incremental compiler design, and the crux of this thesis project as discussed in Chapter 6.

2.2.3. Problems Caused by Names

In the example given in §2.2.1, the *statement* was chosen as the recompilable unit on the basis that a change to a statement could not affect the meaning of any part of the program outside that statement. Unfortunately, the ability to use names in a programming language complicates the task of incremental compilation.

If the part of the source code that is being modified is a declaration then that modification may well affect the meaning of statements throughout the rest of the program. Statements within the scope of the declaration will need to be checked to ensure that the modification to the declaration has not invalidated references to the declared name. If the part of the source code that is being modified is a statement which refers to a name then the validity and meaning of that reference is dependent upon declarations and references elsewhere in the program.

The manner in which various incremental systems have dealt with this problem is discussed in Chapters 3 and 5. The recompilable unit remains (as defined above) that structure which will be recompiled. However, it is important to remember that further checking may be necessary.

---

[3]Note that a *normal* compiler (ie. a "non-incremental" compiler) can be thought of as an α-type compiler with the entire program or (as in the case of Modula-2 or C) a component module as its recompilable unit.

2.3. Development of Incremental Systems

2.3.1. Programming Environments

The idea of building a compiler which compiles incrementally was mooted as long ago as the late 1960s [Braden 68, Katzan 69, Peccoud 69, Rishel 70]. Even so, relatively sophisticated incremental compilers were not implemented until the (fairly recent) development of programming environments. Programming environments use copious amounts of computer resources and it is only with the advent of powerful, single-user computers that the implementation of programming environments has become feasible.

A programming environment provides the user (the programmer) with a number of integrated, interactive tools so that she/he may create, modify, execute and debug a program.[4] If the environment is to be highly interactive then the programmer must be regularly informed of errors in the program and given the opportunity to correct them. In order for program development to be practicable, the compiler must have a fast response time. To ensure a fast response, the compilation should be done incrementally.

The environment should provide more than just a suite of tools which share a common database of information about the program. The various tools should be presented to the programmer as a single tool; there should be no "fire walls" separating the various functions of the environment. The programmer should be able to develop programs within the environment without having to "perform mental context switches" [Delisle 84].

This amalgamation can be achieved by linking the compiler to the editor (as described in §2.3.2), and by allowing debugging commands to be entered using the language which is being supported by the environment.[5] This latter step obviates the need for a programmer to learn a series of special debugging commands, and makes it easier for the programmer to view the environment as a single paradigm.[6]

---

[4]Cedar [Teitelman 84, Swinehart 86] is an example of a *complete* environment; as well as providing a programming environment, facilities exist for document processing, electronic mail and graphics image editing.

[5]For example, the *Interlisp* system [Teitelman 81] provides a single command language for programming, debugging and editing.

[6]The authors of [Delisle 84] make the point that, in such a system, "The debugging mechanisms inherently follow not only the notation and semantics of the programming language, but also its philosophy."

Debugging commands entered in the supported language can be (incrementally) compiled and executed. However, this approach may prove to be disadvantageous in some cases. If the programming language which is supported by the environment is highly-readable but verbose, it will be difficult for the programmer to construct concise debugging commands. The disadvantage of having a verbose debugging language must be balanced against the advantage of allowing the programmer to view the environment as a single paradigm.

2.3.2. Syntax-Directed Editors

A *syntax-directed editor* (or SDE) allows the programmer to edit the program within the context of the language in which that program is being written. Programs are stored internally not as a list of characters but as a parse tree. The program is edited in terms of that parse tree, rather than in terms of the textual representation of the program. This means that the operation of the SDE can be strongly linked with that of an incremental compiler, which is one reason why programming environments usually employ SDEs.

An SDE can be generated from the specifications of a programming language.[7] It is often expedient to modify that specification so that commonly-used constructs can be created in the SDE without having to move through an inordinately large number of levels.[8] Conversely, it is often useful to modify the language specification by adding new levels of structure to save the programmer from being offered a surfeit of choice at each level.

SDEs provide the programmer with two types of command: generic tree manipulation (*e.g.* deleting a sub-tree from the parse tree; traversing a sub-tree), and language specific commands (*e.g.* creating a specific statement). Cursor movement can be *structural* or *textual*. Structural movement is constrained by the structure of the parse tree that represents the program. Although such movement is often sufficient, it can be frustrating for the programmer if the destination is "virtually close but structurally far away" [Garlan 84]. For this reason, most SDEs allow both structural and textual movement.[9]

---

[7]The Cornell Synthesizer Generator [Reps 84] and the PSG system (see §3.3.6) use attribute grammars (see §3.4) to generate syntax-directed editors for arbitrary languages.

[8]Examples of this are given in [Garlan 84].

[9]Textual movement is often implemented using a pointing device (*e.g.* a *mouse*).

2.3.2.1. Advantages and Disadvantages

SDEs simplify the programmers editing task in a number of ways. Keywords can be specified in an abbreviated form. The SDE will be able to determine which keyword is desired from the syntactical context of the cursor position. Alternatively, a list of those keywords which could validly appear at the current cursor position can be displayed (as a menu) and the desired keyword chosen using some pointing device. This feature can help a programmer to learn the rules of the language.

SDEs make large demands upon computer resources, especially on space required to store the program as a parse tree. However, the main disadvantage of SDEs arises from their insistence that the program be consistently correct before and after each editing change. The shortest or most natural sequence of editing commands which change a legal program P_1 into a legal program P_2 may take the source code through a series of invalid programs. If all errors are flagged as they are detected, the programmer is left to distinguish between substantial errors in the program and those transitional errors caused by the editing changes.

One solution to this problem would be to allow the programmer to effectively turn off the error checking mechanism, and to turn it back on when she/he believes that the code is valid again. This approach makes the programming environment less interactive. Some programming environments solve the problem by not allowing the programmer to move the cursor past the first error detected in the code.[10] In this manner the validity of all of the code above the cursor can be guaranteed, although the programmer may be forced to follow a convoluted path of editing commands to change the program.[11]

Another solution is to use *templates*. This means that the SDE can maintain a syntactically valid program, even though some of the constructs may be shells, from which details are missing.

---

[10] *e.g.* the system discussed in [Morris 81].

[11] In such an environment, the only error which need be flagged is the first; subsequent errors will be flagged when the first is corrected. This may seem an inappropriate manner in which to display errors. However, it must be remembered that the first compilers which gave as many error messages as possible were developed at a time when compilers were run in batch queues, and system resources were scarce. Programmers required as many error messages as possible from each attempted compilation. Such considerations are not relevant to the question of when to flag error messages in an interactive, incremental programming environment.

A further difficulty with using SDEs is that the programmer has to adapt herself/himself to entering expressions in a prefix manner. The developers of the GNOME programming environment claim that those students using GNOME who had programming experience found this awkward at first, while those who had no previous programming experience found it easy [Garlan 84].[12]

2.3.2.2. Triggering Recompilation

Given that the aim of an incremental compiler is to update the object code after each change to the program, it follows that recompilation should be triggered by the SDE. It is important to decide exactly what constitutes an editing change.

The SDE will allow the programmer to indicate, in some way, that a change has been made and can now be processed (*e.g.* by typing the *RETURN* key). A β-type incremental compiler will proceed immediately to find the smallest recompilable unit in order to recompile that. Such a prompt response may be premature if the compiler is α-type. It may be that the programmer wants to make two or more changes within the same recompilable unit. The changes are reflected immediately in the SDE's parse tree, but the α-type incremental compiler may be triggered by the SDE only after the programmer has finished making changes within that recompilable unit. This may be when the SDE cursor is moved out of the recompilable unit, or when the programmer chooses a *compile* option.

Implementing such a system requires that a distinction be drawn between the two main tasks of a compiler:

- *syntactic checking* - ensuring that the program (or program fragment) is syntactically correct; and
- *translation* - converting the program (or program fragment) into an executable form.

The syntactic checking is performed by the SDE when it constructs its parse tree. It is the translation phase of compilation which is triggered after the recompilable unit has been edited.

The use of SDEs makes it difficult to postpone syntactic error checking (as discussed in §2.3.2.1) unless it is possible to store syntactically incorrect code in the parse tree (flagged in some way so as to indicate that the code contains syntax errors). Static semantic error checking can easily be postponed until translation.

---

[12] See also [Chandhok 85].

There is a sense in which this approach departs from the ideal of incremental compilation. After all, the compiler is no longer providing a compiled version after each editing change to the source code.[13] However, such a system remains incremental insofar as it does not require complete recompilation after modifications have been made to a program. It also has the advantage of delaying error checking, effectively turning error checking off until the recompilable unit has been edited.

This approach is adopted in the MAGPIE system (see §3.3.5) and forms the basis of the modifications made to the PECAN system as part of this thesis project (as described in Chapter 6).

[13] Unless one takes the somewhat tenuous view that several editing changes within the one recompilable unit constitute a *single* editing change.

Chapter 3

Examples of Incremental Systems

3.1. Early Incremental Systems

3.1.1. Incremental BASIC - 1968

An implementation of an incremental system for the BASIC language is described in [Braden 68]. This system uses α-type incremental compilation. As each line of code is entered, it is compiled into machine code and a reference to that code is stored in a *program vector*. When a line is modified it is recompiled. Most statements are executed in machine code, but statement-to-statement code[1] is handled interpretively, by moving through the program vector.

There are difficulties in implementing such a system even for a language as context-independent as BASIC. For example, if the user enters the following lines

```
100 DIM X(10)
200 LET X(1)=0
100 DIM X(10,10)
```

the assignment statement in line 200 was valid when first entered but, due to the change in the definition of the X array, it has become invalid. Yet, the system will not recompile the offending line because it was valid when first entered. If the compiler was forced to compile the entire source file in order to rectify this problem then any time saved due to incremental compilation would be lost. One solution would be to treat a reference to an element of a one-dimensional array as a special case of a reference to an element of a two-dimensional array. This would mean that the code generated when line 200 is first entered will still work correctly after the X array is redefined. The authors of [Braden 68] give this solution serious consideration, rejecting it only because it is not sufficiently general to handle all such problems.

The only remaining solution is to recompile only the statement that was changed and check references to the X array for validity at run-time. This solution moves

---

[1] *i.e.* branching statements (GOTO, GOSUB).

the implementation a little away from the ideal of an incremental compiler because the context-sensitive checking is being deferred from compile-time to run-time. But the authors justify using this solution on the grounds that it is preferable to the other options and that the system is intended for use by students who will usually write small programs that are run correctly only once.

3.1.2. Languages with Nested Statements - 1972

Earley and Caizergues describe another α-type incremental compilation system in [Earley 72]. The authors make the point that it is a relatively easy task to incrementally compile programs which have been written in a language which does not allow nested statements. In such a language the meaning of each statement is usually independent of those statements around it, so it is necessary to recompile only the lines that are actually altered. If a declaration is changed, the recompilation can be limited to those statements within the scope of the declaration. However, if the language allows nested statements then the question of statement independence can be greatly complicated.

The authors' solution to this problem is to distinguish between simple and nested statements. The language is redefined so that single statements may only appear on a single line, while nested statements may appear on several lines. *Skeleton entries* are maintained for each line of code. These entries link the source line with the corresponding compiled code and each includes a pointer to the next line's skeleton entry. If the line is the beginning of a nested statement, a pointer in the skeleton entry refers to the entry for the line which ends the nested statement. If part of a nested statement is modified, only the body of that nested statement need be recompiled. Although the authors see the structure as a list of statements, the skeleton entries could just as easily have been thought of as nodes of a tree.[2]

The authors identify a problem with this method where the language being implemented does not have an explicit *end* for each nested statement. However, it would seem that such languages could be implemented simply by defining an *end* (with a null production) for each nested statement.

The appropriate lines are recompiled only when all of the editing is complete. This delay is for two reasons: it avoids duplicating recompilation, and it doesn't force the user to keep the source code syntactically correct at all times.

---

[2]Indeed it is difficult to see why a tree structure was not used; it would seem to be a preferable paradigm.

3.2. Conversational Systems

Conversational systems were precursors of the more sophisticated incremental compilers. A conversational system can be distinguished from a system which incorporates incremental compilation by the fact that, although it aims to provide a high level of interactivity, it still compiles *all* of the source code when changes are made.

3.2.1. CONA and COPAS - 1978 and 1981

The CONA and COPAS systems [Atkinson 78, Atkinson 81a] are implementations of conversational Algol and conversational Pascal respectively. The program's source code is converted into an intermediate form which can be efficiently interpreted. When changes are made to the program, the entire program (that is the intermediate representation and the new text) is converted into the intermediate form. Modifications to the code are checked for validity immediately. If the source contains an error, the compiler halts and waits until the error is corrected before the rest of the text is scanned.

Neither of these systems is significantly faster than a system which has a separate text editor and compiler, but the designers point out that the conversational systems were designed for use by novices who write small programs. For small programs this method compiles code quickly enough, and both systems do provide the user with recompiled code after each modification.

3.3. Incremental Systems in Programming Environments

3.3.1. The Cornell Program Synthesizer - 1978

The Cornell Program Synthesizer [Teitelbaum 81] was the first major programming environment to treat programs as "a hierarchical composition of syntactic objects, rather than (as) a sequence of characters." The Synthesizer supports the development of programs in PL/CS (a dialect of PL/I). Programs are edited using an SDE. Templates are used for all but the lowest level language structures (or *phrases*) which are entered as a character string and parsed. Phrases are checked for syntactic and semantic errors. Compilation (into an interpretable form) is performed each time a template or phrase is inserted.

Incomplete programs may be executed. Execution halts when an unfilled template is encountered, but can be resumed after editing changes have been made (unless a declaration is altered). If a change is made to a declaration, all of the phrases within the scope of that declaration are re-checked.

The Synthesizer has been generalized with the development of the Synthesizer Generator [Reps 84] which generates SDEs from languages specified using attribute grammars.

3.3.2. Smalltalk-80 - 1980

Smalltalk-80 [Goldberg 83, Goldberg 84] is an interactive, integrated programming environment. Smalltalk-80 is also an object-oriented programming language supported by the Smalltalk-80 environment. The environment is defined in terms of the language so the programmer is presented with a single paradigm.

The basic element in the Smalltalk-80 language is the *object*, which has its own data (not accessible by other objects) and *methods*. Methods are programs which respond to messages passed between objects. Programming in Smalltalk-80 is a matter of creating objects and specifying how those objects will communicate with each other. Methods are edited using a simple text editor. Smalltalk-80 uses α-type incremental compilation, using the *method* as the recompilable unit. Methods are translated into sequences of instructions for a stack-oriented interpreter.

3.3.3. IPE - 1981

The IPE (Incremental Programming Environment) system is described in [Medina-Mora 81]. IPE supports the development of programs in the language GC (a variant of the language C, with module structure and type checking). Programs are edited using a SDE which is completely template-driven; textual input is not supported. The editor ensures syntactic correctness and performs semantic checking.

IPE uses an α-type incremental compilation strategy. Only when a procedure is semantically correct, is code produced. The procedure is automatically compiled, loaded and linked into the existing executable code for the program. If a subsequent change outside the procedure (*e.g.* to the declaration of another procedure) makes an already compiled procedure semantically incorrect, that procedure code is replaced by a code stub. If executing the program causes that code stub to be executed (*i.e.* if the semantically incorrect procedure is invoked) then execution halts so that the procedure may be modified.

IPE was designed "to provide the comfort of a flexible and interactive programming environment for compiler-based languages." To this end it maintains two internal representations of the program under development: the tree

representation and the executable representation. The executable representation is generated from the tree representation, and may be generated so that it can be executed on a different system from that on which the IPE system is being run.

3.3.4. PECAN - 1984

The PECAN programming environment generator is discussed, in considerable detail, in Chapter 4.

3.3.5. Magpie - 1984

The Magpie programming environment supports the development of Pascal programs on an experimental workstation. The system's method of incremental compilation is described in [Schwartz 84]. Magpie uses a sophisticated α-type compilation technique.

Magpie divides Pascal programs into fragments: statement bodies, variable declarations, constant definitions, type definitions, label declarations and headings (of procedures, functions and the main program). The text of these fragments is stored as a sequence of tokens. Use of an *uninterpreted* token (representing an incomplete token, an incorrect token or un-scanned text) means that all of the text can be tokenized at any time.

Magpie breaks the compilation process into three distinct phases: scanning, parsing and recompilation (translation into machine code). Each of these phases has its own unit of incrementality. Scanning will respond to a changed character, but the parser will not respond to that change unless it means a change to a token. For example, changing the value of an integer constant means only a small change to the appropriate token. However, if the change to the text changes the type of the token (say, from an integer constant to a real constant) then the parser is invoked.

Any single change to the source code is bounded by a single fragment, not by the entire text, so the parser can confine itself to that fragment. Each fragment is edited separately, and has its own cursor. Magpie uses a textual editor. This precludes static semantic checking beyond the first syntax error within each fragment. The syntactic structure of each fragment is maintained as a sequence of partial parse trees.

Recompilation is performed on a procedure-by-procedure basis, and is triggered when a cursor leaves a fragment. Recompilation of a procedure is performed in

the background when the processor is not busy providing the programmer with interactive response.[3] If execution commences before all of the compilation has been finished then Magpie executes the existing code, pausing to generate code for uncompiled procedures that are invoked during program execution.

Magpie uses Pascal as a debugging language. The programmer is able to invoke code in a given activation record, and to define *demons* (procedures that can be set up so that they are invoked whenever reference is made to a specified identifier). These demons can be disabled, although the "hook" into the compiled code remains.

3.3.6. PSG - 1986

The PSG programming system generator is described in [Bahlke 86]. It produces programming environments for a language given a definition of the language specified using an attribute grammar (see §3.4).

The language definition is divided into three parts:

- syntax
- context conditions (scope and visibility rules, data attribute grammar, basic context relations)
- dynamic semantics (domain definitions, auxiliary functions, meaning of executable parts of program, meaning functions).

The syntax of the language is mandatory. If the context conditions are not specified then the editor which is generated will be context-free. If the dynamic semantics are not defined then the environment which is generated will have no means of compiling programs written in that language.

The editor that is generated allows both structure editing and text editing. Where structure editing is used, the programmer is only given menu options which are syntactically and semantically valid. Hence the editor can guarantee the prevention of syntax errors and semantic errors. When textual editing is used, such errors will be recognized immediately and flagged, but not prevented.

Programs are interpreted using the dynamic semantics information provided. Incomplete programs can be interpreted until an attempt is made to interpret a syntactically incomplete structure. The PSG system has been used to produce

---

[3] During the programmer's "think time" (*sic*) [Delisle 84].

environments for Pascal, Algol-60, Modula-2 and for its own formal language definition language.

3.4. Attribute Grammars and Environment Generators

An attribute grammar[4] is a context-free grammar which has been augmented with information which specifies context-dependent aspects of the language. Trees generated from attribute grammars are called *attributed structure trees*. Each node of a structure tree has an associated *attribute* which describes properties of that node.

Attribute grammars have been used in parser generating systems[5] and to generate SDEs.[6] As explained in §3.3.6, the PSG system can generate an entire programming environment for a language specified using an attribute grammar. However, there are several drawbacks associated with using attribute grammars in generator systems.

Specifying a language using an attribute grammar requires that a substantial number of functions be specifically designed for that specification. These functions provide the language's semantics, and the attribute grammar provides the dependency information used when finding the smallest recompilable unit. This dichotomy between semantics and dependency information adds to the complexity of a language specification. Language specification in PECAN (see §4.1.2) uses a specification language which provides dependency information and (almost) all the semantic information without recourse to additional functions.

If a language specification is based upon an attribute grammar, the symbol table is usually represented by a set of state variables at each node of the structure tree. This has the inherent disadvantage that a large part of the program has to be recompiled whenever a change is made to a declaration. PECAN avoids this problem by determining exactly what references are affected by a change to a declaration, and processing only those references.

---

[4] For a comprehensive discussion of attribute grammars see Chapter 8 of [Waite 84].

[5] *e.g.* GAG [Kastens 82].

[6] *e.g.* (as already mentioned) the Cornell Synthesizer Generator [Reps 84].

Chapter 4
The PECAN Programming Environment Generator

4.1. Introduction

The PECAN programming environment generator was developed at Brown University, Providence, U.S.A., under the direction of Steven Reiss. It is a large collection of large modules written in the C programming language and executable under the UNIX operating system. PECAN was initially designed to run on Apollo workstations, but has been adapted for use on Sun workstations.[1]

4.1.1. Documentation

The PECAN system is very poorly documented. Although a user guide exists [Barlow 86a], there is little information available about the internal workings and structure of PECAN. Apart from a few papers on PECAN's component modules, the main sources of information are [Reiss 83, Reiss 84a, Reiss 84b].

Various aspects of the system are discussed in [Barlow 86b, Leung 86, Nearhos 86, Purdue 86]. This relative dearth of information about the PECAN system leaves anyone interested in its workings with no choice but to examine the code. Unfortunately, the internal documentation is terse, bordering on the Trappist.

4.1.2. Language Specification

PECAN is a programming environment generator. A language's syntax and semantics are specified in PECAN's own high-level specification language.[2,3] PECAN produces language-specific code from the specifications, which is merged with existing language-independent modules to form code which provides the programming environment.

---

[1]The project that is the subject of this thesis was developed using PECAN on a Sun-2 workstation at the Computer Science Department, Australian National University.

[2]PECAN does not use attribute grammars to specify languages for the reasons given in §3.4.

[3]The specification of the Pascal WHILE statement is given in Figure 5-2.

The specification of a language is broken into four parts:

- an abstract syntax of the language and the semantics of each construct in the language;
- the properties of its symbols;
- a definition of the types allowed in the language, and details of type coercions for resolving expressions; and
- details of how to build and resolve expressions.

Theoretically, PECAN can generate an environment for any language that is algorithmic, block-structured and makes no explicit use of parallel processing. However, an extended version of Pascal (based on [Jensen 78]) is the only sophisticated programming language for which a reasonable environment has been generated. An environment for the mini-language Core (as defined in [Ledgard 81]) has been generated, but the language Modula-2 [Wirth 83] proved too complicated for one honours student in 1986 [Leung 86]. The specification for Pascal is some 4000 lines, and a language as simple as Core required about 1200 lines to be specified for PECAN. It can be seen that the specification of a language for PECAN is a complicated task.

So, although PECAN is an environment generator, the only practical and useful environment which has been generated is that for Pascal. Future references to "PECAN" in this thesis will be references to the environment generated by the PECAN programming environment generator for the language Pascal.

4.1.3. Views

PECAN makes good use of the graphical capacity of the Apollo and Sun workstations, providing the programmer with many views of the program under development; multiple views of the shared data structures of PECAN's various component modules. These views can be divided into five categories:[4]

- Program Views
  - Syntax-Directed Editor (SDE module - see §4.1.3.1)
  - Nassi-Schneiderman View (NASSI module)
  - Declaration View (DECL module)
  - Box Editor
  - Rothon Editor
- Semantic Views (static semantic meaning)
  - Symbol Table View (SYMMOD module)
  - Data Type View (TYPE module)
  - Expression View (EXPR module)
  - Flow View (FLOW module - see §4.1.3.2)
- Execution Views (dynamic semantic behaviour)
  - Interpreter View (PALM module)
  - Stack View (STACK module)
- System Views
  - Transcript View (CMD module)
- Miscellaneous Views
  - Draw Window
  - Clock Window
  - Button Window
  - Pics Window

---

[4]Roughly corresponding to the division in [Reiss 84b].

All views provide up-to-date information on the state of the program or of its execution. When changes are made in one view, that change is reflected immediately in all other appropriate views. For example, if a change is made in the SDE then that change is immediately reflected in the other program views. The various semantic views will reflect the change if it is relevant (*e.g.* if a change is made to a statement, that change is reflected in the flow view; if a change is made to an expression, that change is reflected in the expression view).

An example PECAN screen is given in Figure 4-1. The screen shows several views of a program[5] which was in the process of calculating the value of 7!, before execution was halted. The views shown are (clockwise from the top left) the syntax-directed editor, the symbol table view, the clock window, the flow view, the stack view, the expression view, the transcript view, and the interpreter view.

4.1.3.1. The Syntax-Directed Editor

Program views provide the programmer with a visual representation of the abstract syntax tree (discussed in §4.2.2). The SDE allows both structural and textual cursor movement. Furthermore, the programmer may move the cursor directly to any part of the program using the pointing device. The programmer may use templates to build a program using menus to choose keywords and constructs. Alternatively, text may be entered and will be parsed (one line at a time).[6] All errors are flagged when detected.

4.1.3.2. The Flow View

The flow view represents the program in flow chart form. Flow charts are constructed using a differently-shaped box to represent each of the following structures: the start; a variable declaration; a statement; a condition; an entry or exit point into a procedure or function; a junction of paths; and the end.

The flow view's cursor responds to changes in other views, and if a node in the flow graph is chosen (*i.e.* pointed to) then other program views will reflect the change. This is the extent of interactivity allowed in the flow view.

---

[5]The test program *test3.p* (see §C.3).

[6]PECAN uses a parser based upon Earley's parsing algorithm. A detailed description of Earley's algorithm is given in Appendix D.

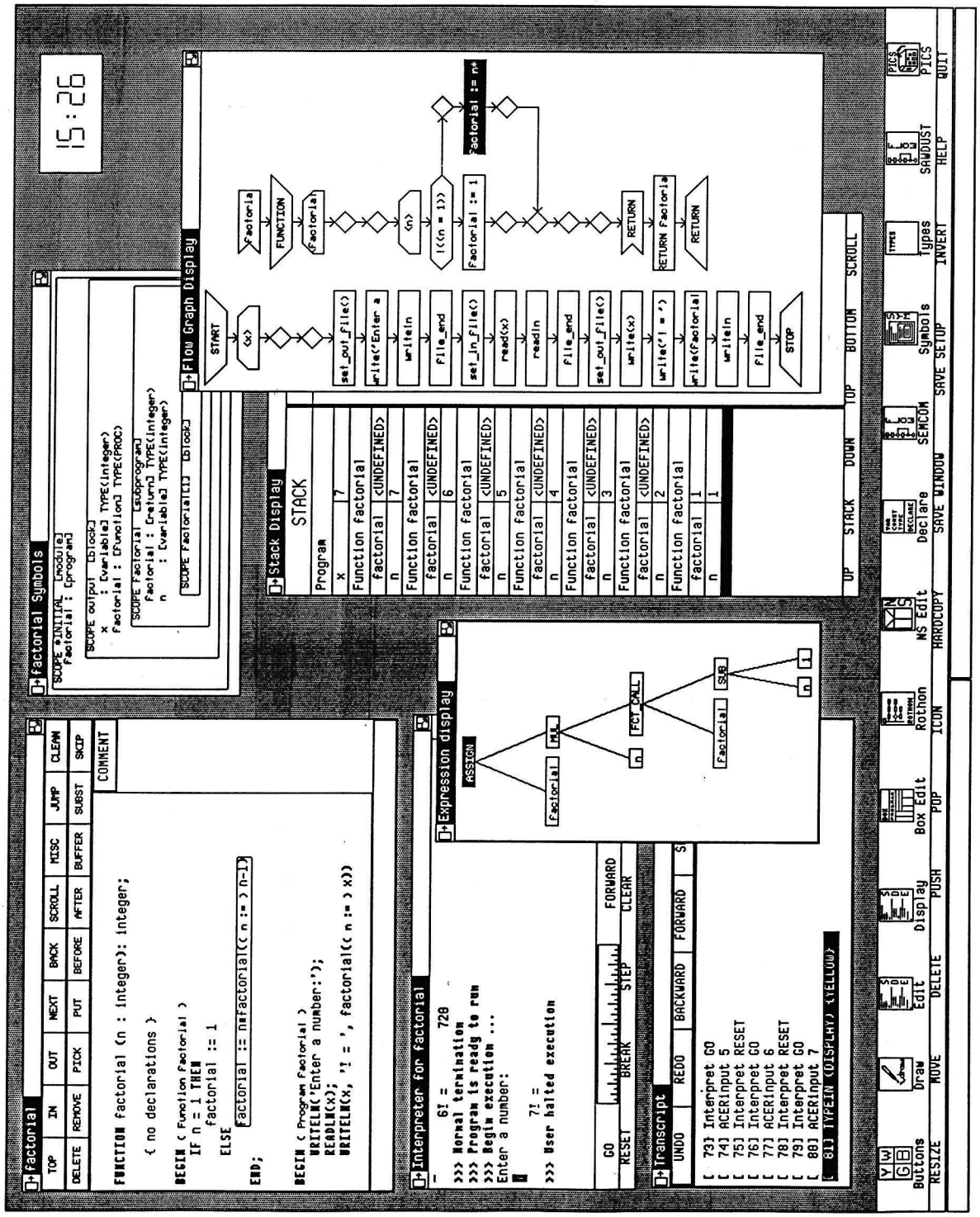

Figure 4-1: PECAN Views

4.2. Internal Structure

4.2.1. Modules

PECAN has a hierarchical module structure. This reflects the fact that PECAN was developed to work in an existing environment: the Brown Workstation Environment [Bazik 85]. The hierarchy of modules is shown in Figure 4-2.

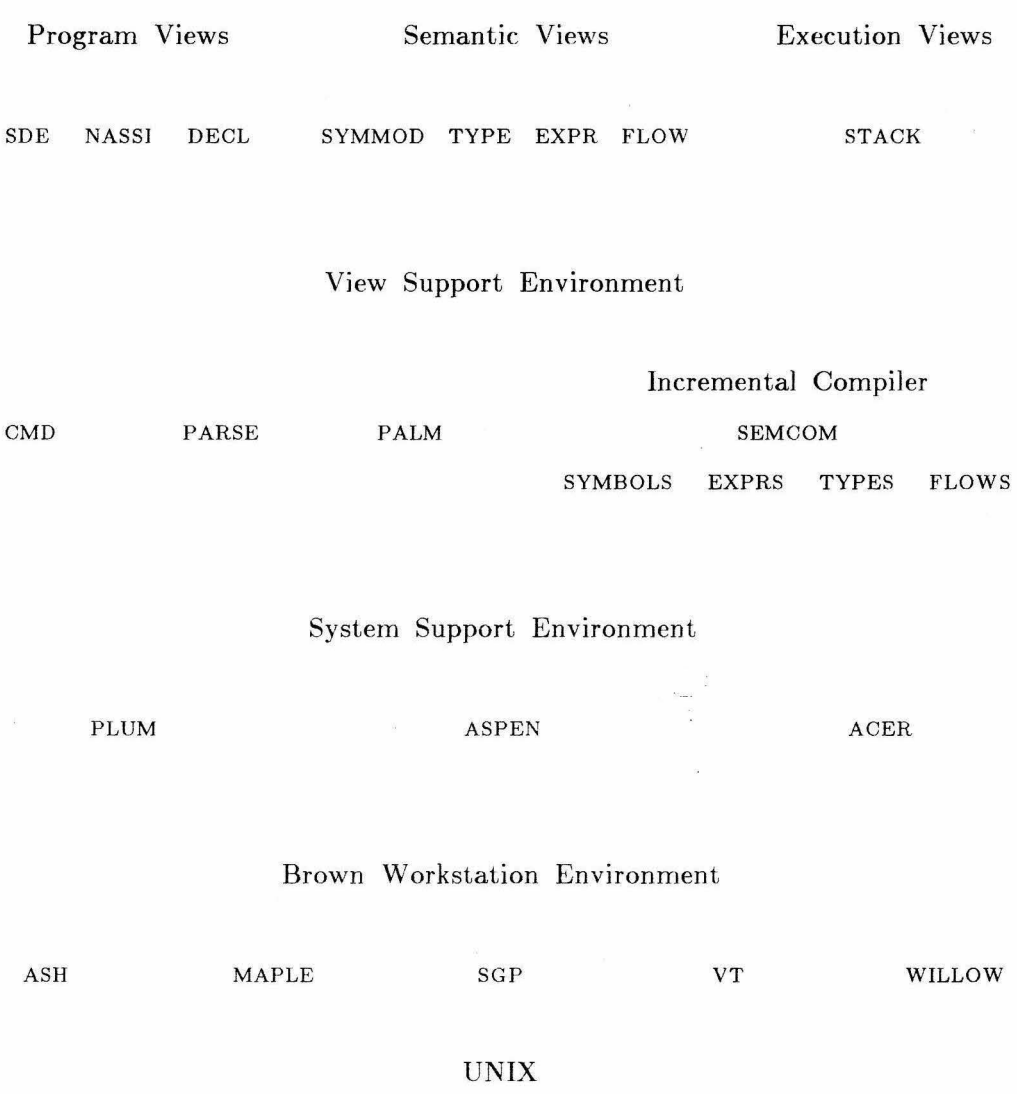

Figure 4-2: Hierarchy of Modules in PECAN[7]

[7] Adapted from a figure in [Nearhos 86].

Several of the modules provide an abstract data type (with its own data structure and operations) to the other modules. The module with which this thesis is primarily concerned is the SEMCOM module. The operation of SEMCOM is discussed in detail in Chapter 5.

4.2.2. The Abstract Syntax Tree

The main data structure which is used by all modules is the Abstract Syntax Tree (or AST). The AST is supported by the ASPEN module [Molinari 86]. As well as maintaining information about the structure of the program, the AST provides links to data structures used by other modules. Thus, the AST is the central data structure; access to all other data structures can be gained (perhaps indirectly) through the AST.

4.2.3. Events

In order for PECAN to present the programmer with an integrated environment, it is essential that the various modules have a means of communicating with each other. For example, a change made to the program in the SDE may have effects upon all other views. It is clearly undesirable that any one module should have to explicitly invoke functions in other modules in order to propagate a change throughout the system. As well as being cumbersome to code, such an approach makes future expansion of the system very complicated. PECAN solves the problem of module communication by use of *events*.

An event is effectively an announcement by one module, to any other module that might be interested, that some specified happening has occurred. Events are broadcast by the PLUM module [Molinari 85].[8]

The event structure is set up in the following manner. When PECAN is first invoked, the main program calls the initialization functions for each module. Each module's initialization function registers (with PLUM) the events in which the module has an interest. This expression of interest is made using the *PLUMaccept_event* function. *PLUMaccept_event* takes two arguments: a function in the interested module, and the name of the relevant event. Any number of modules may register an interest in a given event.

---

[8] Note that although events are broadcast, execution is sequential; concurrent execution is not supported.

The PLUM module maintains a list of functions registered for each event. When a module wishes to trigger an event, the *PLUMevent* function is used. PLUM invokes, in turn, each of the functions linked to that event. Parameters may be passed to the *PLUMevent* function. These parameters are passed to the interested functions when an event is propagated throughout the system.

Chapter 5
Incremental Compilation in PECAN

5.1. Semantic Specification Statements

The PECAN approach to incremental compilation is described (somewhat inaccurately[1]) in [Reiss 84a]. The SEMCOM module handles incremental compilation in PECAN. To achieve this, SEMCOM maintains its own language-independent representation of the semantic meaning of the AST - a list of statements in a simple semantic language. These statements are referred to as *semantic specification statements.* A brief description of the meaning of each of these statements is given in Figure 5-1.

These statements can be divided into two categories: *action* statements and *control* statements. When they are *executed*, action statements build the underlying representation of the program. This underlying representation forms the data structure used by the flow view to display the program in flow graph form. This flow graph representation is directly interpreted when the program is run. Control statements specify the order in which the action statements are executed. The language uses a stack and a small set of variables called *current items.* The current items are:

- the current scope;
- the current referenced object;
- the current flow graph node;
- the current type;
- the current expression;
- the current auxiliary scope;
- the last type built; and
- the current mode.

---

[1] See second footnote on page 41.

DO	Visit a specified sub-tree.
FOR	Visit each of the children of a list-type node.
START	Create an *INITIAL* scope (marks the beginning of the tree walk).
BEGIN	Create a new scope.
END	Close the current scope, and return to the parent scope.
FIND	Find the symbol table name associated with the specified string or token.
LOOK	Partially resolve a name given specified restrictions.
USE	Resolve a name to a single object.
BUILD	Create a new object of a given type.
DEFINE	Take a newly created object and associate it with the current name.
SET	Set the current symbol.
GET	Access the current symbol.
VALUE	Determine the value of a constant given its textual representation.
MODE	Set flags that affect the current symbol's storage class, and the type of parameter that it may represent (*inter alia*).
PUSH	Push current symbol onto the stack.
POP	Pop current symbol off the stack.
EXPR	Build an expression from the top elements of the stack (using the current symbol as an operator, with a specified number of operands.
FLOW	Attach a new node to the flow graph representation.
TYPE	Build a data type.
CLEAR	Initialize the current items.

Figure 5-1: Semantic Specification Statements[2]

[2]Adapted from [Reiss 84a] and [Molinari 87a].

Semantic specification statements make use of the current items in order to reflect the semantics of each construct in the programming language. Information is passed between semantic statements via the current items. The main advantage of this approach is that it becomes possible to extract dependency information from the specification of each construct, in order to determine the smallest recompilable unit.

5.2. Specifying a Construct

The sequence of semantic specification statements associated with each construct in the programming language forms part of the language specification (discussed in §4.1.2). The specification of the Pascal WHILE statement is given in Figure 5-2. This specification can be thought of as a set of instructions to PECAN as to how to "compile" a WHILE statement.

```
STATEMENT ::= while_statement;

while_statement =>                IF_EXPRESSION STATEMENT ::
        SOURCE: "WHILE @1 DO@+@R@c@n@2@-"
        COMMENT
        SYNONYM: "While"
        SEMANTICS:{     CLEAR;
                        BEGIN loop;
                        DEFINE NAME=operator,EXIT,CLASS=label;
                        DEFINE NAME=operator,NEXT,CLASS=label;
                        USE NAME=operator,NEXT,CURRENT=ONLY;
                        FLOW LABEL=1,LABEL=REF;
                        DO @1;
                        FLOW NOTTEST,2;
                        DO @2;
                        FLOW GOTO=1;
                        USE NAME=operator,EXIT,CURRENT=ONLY;
                        FLOW LABEL=2,LABEL=REF;
                        END;
                  }
        SEEDY: "WHILE @~ @n X1 @' WBLOCK @~ @n @2 @' @n WEND"
        ROTHON: LOOP @1 : @2 : ;
        NS: LOOP @1 @2 NONE;
        ;
```

Figure 5-2: Specification of Pascal WHILE Statement[3]

The string labelled SOURCE is used by the parser, and by the SDE for formatting the construct. COMMENT indicates that a comment may be attached to the WHILE statement. The SYNONYM is the name of the construct for use by the SDE in creating menus for template selection.

[3]This specification of the WHILE statement is taken from the specification used to generate a PECAN environment for Pascal at the Australian National University. It differs slightly from the specification given in [Reiss 84a].

ROTHON and NS define the representation of the WHILE statement for the Rothon editor and the Nassi-Schneiderman view respectively. SEEDY defines the representation for an apparently unimplemented view.

The statements between the curly brackets labelled SEMANTICS are the semantic specification statements for the WHILE statement. The CLEAR statement initializes the current items. This states that the WHILE statement is completely independent of preceding Pascal statements. The BEGIN statement starts a scope of type *loop*. The two DEFINE statements define an *EXIT* label and a *NEXT* label in the operator auxiliary table. The USE statement extracts the *NEXT* label for use in the subsequent FLOW statement. The FLOW statement defines two labels in the flow graph: *NEXT* and a temporary label 1. The DO statement causes the semantic specification statements associated with the IF_EXPRESSION sub-tree to be processed next. The FLOW statement causes a jump to temporary label 2, if evaluating the IF_EXPRESSION returns false. The second DO statement processes the body of the WHILE statement, and the third FLOW statement causes an unconditional branch back to temporary label 1. The USE statement and the FLOW statement access, and attach to the flow graph, the *EXIT* label and temporary label 2. The END statement ends the *loop* scope which was begun with the BEGIN statement.

5.3. Data Structure

5.3.1. SEMCOM_STMTs and the Abstract Syntax Tree

SEMCOM stores its semantic specification statements as a doubly-linked list of record structures called **SEMCOM_STMTs**.[4] Each of these SEMCOM_STMTs contains:

- pointers forwards and backwards to other SEMCOM_STMTs (used to maintain the doubly-linked list);
- details of the type of semantic specification statement being represented;
- a pointer into the AST (for arguments to the semantic specification statement); and
- the values of the current items.[5]

The semantics of the entire program can be represented by a list of SEMCOM_STMTs. Each node of the AST has a pair of pointers which mark the beginning and the end of the list of SEMCOM_STMTs which give the semantics of the construct at that particular node. This is illustrated in Figure 5-4.

5.3.2. SEMCOM_STMTs and the Flow Graph Representation

Consider the Pascal program listed in Figure 5-3. Using the specification of the WHILE statement (given in Figure 5-2), PECAN parses the WHILE statement into a tree (shown in Figure 5-5). The SEMCOM module produces a list of SEMCOM_STMTs which give the semantics of that particular instance of the WHILE statement. The list of SEMCOM_STMTs produced for this example appears in Figure 5-6. The beginning and the end of each of the sub-lists of the list are labelled with the name of the associated node of the tree.[7] When this list of SEMCOM_STMTs is executed, the flow graph representation of the WHILE statement is constructed. The flow graph representation for this WHILE statement appears in Figure 5-7.

---

[4]Note that the mapping from semantic specification statements to SEMCOM_STMTs is not quite one-to-one. Each action statement in the semantic specification is mapped into one or more SEMCOM_STMTs. Statements like USE and LOOK can imply several actions, and the interpretation of statements like SET can depend upon their arguments.

[5]One of the current items is the current flow node. It is through this pointer that the associated (interpretable) flow graph representation of the program is accessed.

[7]Part of this thesis project involved the development of a new PECAN view which displays the SEMCOM_STMTs associated with the current node (as indicated by the cursor in the SDE or some other program view). The list in Figure 5-6 was prepared using this semantic actions view. Details of this new view are given in Appendix A. The form in which SEMCOM_STMTs are displayed is explained in §A.1.

```
PROGRAM interminable (input,output);

    { no declarations }

BEGIN { Program interminable }
    WHILE true DO
        WRITELN('loop');

END.
```

Figure 5-3: Small Pascal Program with an Example WHILE statement[6]

5.4. Execution and Unexecution

When a sequence of SEMCOM_STMTs is *executed*,[8] a flow graph representation is constructed. This flow graph representation is interpreted in order to *run* the program. SEMCOM_STMTs can also be *unexecuted*. Unexecuting a sequence of SEMCOM_STMTs has the effect of removing, from the flow graph representation, those constructs which were created when that same sequence of SEMCOM_STMTs was executed.[9]

This symmetry of SEMCOM_STMTs - the fact that they can be both executed and unexecuted - is essential to PECAN's approach to incremental compilation. Ignoring (for the moment) the problems involved in finding the smallest recompilable unit, the process of incremental compilation can be thought of in the following manner. When a node is changed in the AST, the SEMCOM_STMTs associated with the old node are unexecuted. This has the effect of removing, from the flow graph, the code corresponding to the node as it was before alteration. Next, the SEMCOM_STMTs associated with the new AST node are executed. This inserts, into the flow graph, the code corresponding to the new node. The flow graph is now, as before, an interpretable representation of the program (as amended).

---

[6]Note that this program listing was formatted by PECAN, using the formatting information included in the specification of Pascal.

[8]The execution of SEMCOM_STMTs should not be confused with the execution of the program (*i.e.* the interpretation of the flow graph representation).

[9]The functions that perform execution and unexecution consult and update the values of the current items, as discussed in §5.5.2.5.

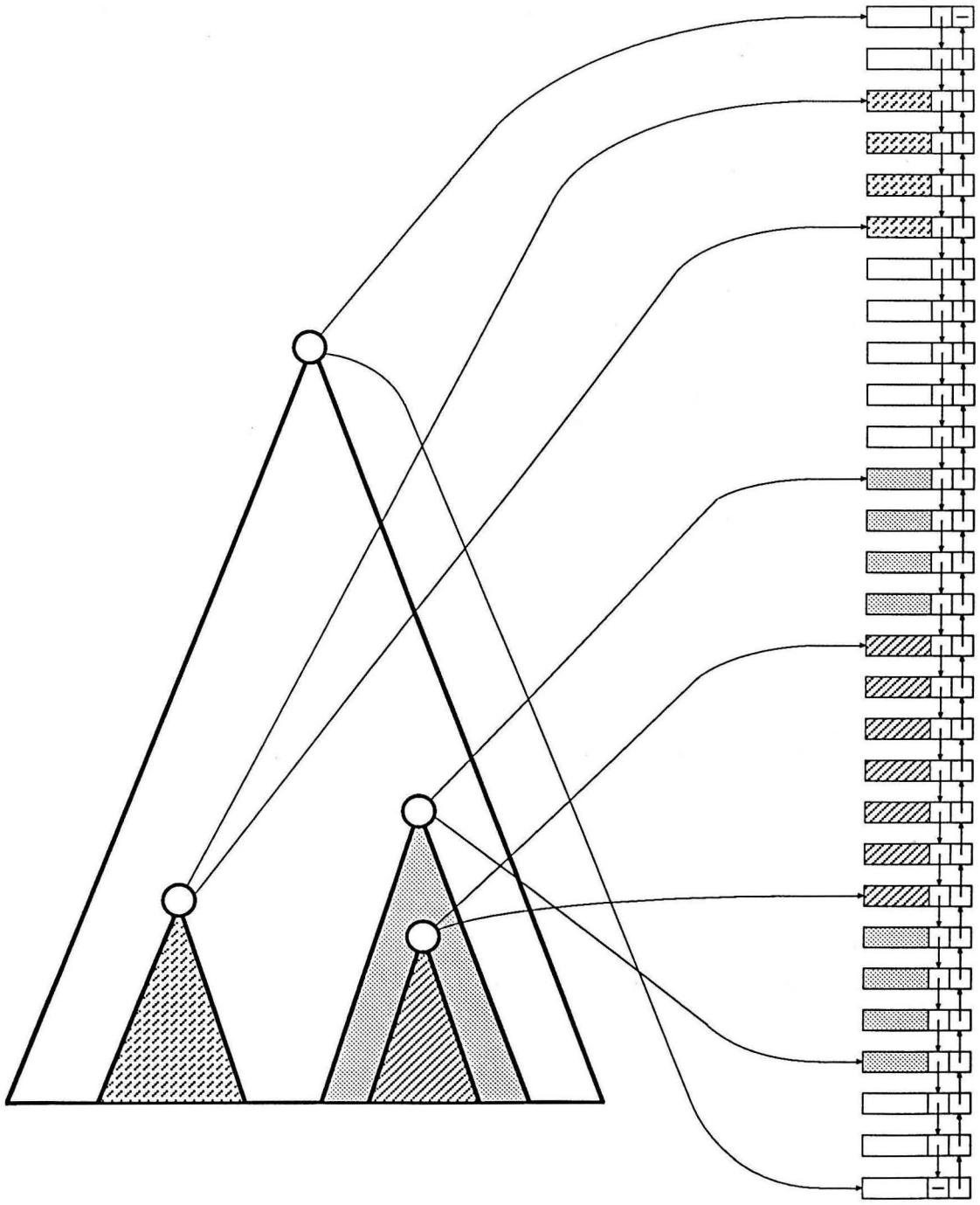

Figure 5-4: Abstract Syntax Tree with Pointers into List of SEMCOM_STMTs

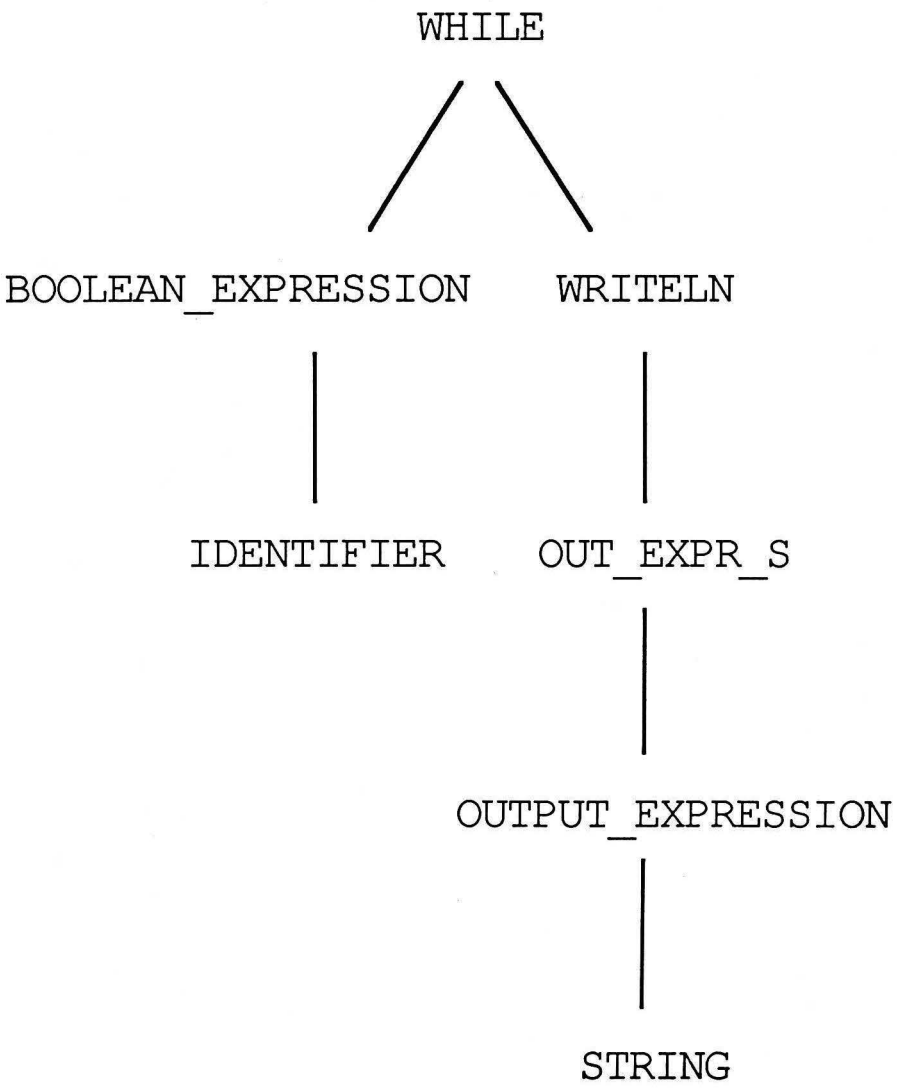

Figure 5-5: Parse Tree for Example WHILE Statement

```
WHILE (begin)
   (0x2484f8) :    SAVE_CUR 0                [0x2486c8]      @0x24a3e8
   (0x24dad0) :    FLOW_ALLOC 519            [0x248ae0]      @0x24a3e8
   (0x24dab4) :    CLEAR 521                 [0x0]           @0x24a3e8
   (0x24da98) :    BEGIN 522                 [0x218964]      @0x24a3e8
   (0x24da60) :    CHECK_LEX 0               [0x0]           @0x24a3e8
   (0x24da7c) :    FIND 524                  [0x21b3e4]      @0x24a3e8
   (0x24da44) :    BUILD 527                 [0x24929c]      @0x24a3e8
   (0x24da28) :    DEFINE 529                [0x21b3e4]      @0x24a3e8
   (0x24d9f0) :    CHECK_LEX 0               [0x0]           @0x24a3e8
   (0x24da0c) :    FIND 531                  [0x249494]      @0x24a3e8
   (0x24dd90) :    BUILD 534                 [0x249260]      @0x24a3e8
   (0x24dd74) :    DEFINE 536                [0x249494]      @0x24a3e8
   (0x24dd3c) :    CHECK_LEX 0               [0x0]           @0x24a3e8
   (0x24dd58) :    FIND 538                  [0x24946c]      @0x24a3e8
   (0x24dd20) :    R_CURONLY 541             [0x21b5f4]      @0x24a3e8
   (0x24dd04) :    USE 542                   [0x24946c]      @0x24a3e8
   (0x24dce8) :    FLOW_BLOCK 543            [0x248aac]      @0x24a3e8
   (0x24dccc) :    FLOW_BLOCK 546            [0x248a78]      @0x24a3e8
   (0x24dcb0) :    SYMBOL_SET_FLOW 549       [0x24946c]      @0x24a3e8
BOOLEAN_EXPRESSION (begin)
   (0x24df1c) :    SAVE_CUR 0                [0x2486a0]      @0x24a520
IDENTIFIER (begin)
   (0x24dbec) :    SAVE_CUR 0                [0x248678]      @0x24a554
   (0x24dc78) :    CHECK_LEX 0               [0x218d82]      @0x24a588
   (0x24dc94) :    FIND 1604                 [0x249444]      @0x24a588
   (0x24dc5c) :    R_CLASS 1843              [0x248dfc]      @0x24a554
   (0x24dc40) :    R_CURNEST 1853            [0x21b5e4]      @0x24a554
   (0x24dc24) :    USE 1854                  [0x249444]      @0x24a554
   (0x24dc08) :    EXPR_REF 1855             [0x20cad0]      @0x24a554
IDENTIFIER (end)
   (0x24df70) :    CHECK_LEX 0               [0x0]           @0x24a520
   (0x24dbd0) :    FIND 1327                 [0x24941c]      @0x24a520
   (0x24df54) :    USE 1330                  [0x24941c]      @0x24a520
   (0x24df38) :    EXPR_BUILD 1331           [0x20ca98]      @0x24a520
BOOLEAN_EXPRESSION (end)
   (0x24df00) :    FLOW_BLOCK 552            [0x248a44]      @0x24a3e8
WRITELN (begin)
   (0x24e218) :    SAVE_CUR 0                [0x248650]      @0x24a41c
   (0x24dee4) :    CLEAR 1123                [0x0]           @0x24a41c
   (0x24deac) :    CHECK_LEX 0               [0x0]           @0x24a41c
   (0x24dec8) :    FIND 1124                 [0x2493f4]      @0x24a41c
   (0x24de90) :    EXPR_BUILD 1127           [0x20ca60]      @0x24a41c
   (0x24de74) :    FLOW_BLOCK 1129           [0x248a10]      @0x24a41c
OUT_EXPR_S (begin)
   (0x24e314) :    NOP 0                     [0x0]           @0x24a450
OUTPUT_EXPRESSION (begin)
   (0x24df90) :    SAVE_CUR 0                [0x248628]      @0x24a484
   (0x24de58) :    EXPR_BEGIN 1151           [0x20ca28]      @0x24a484
```

Figure 5-6: List of SEMCOM_STMTs for Example WHILE Statement

(continued next page)

```
STRING (begin)
   (0x24e01c) :    SAVE_CUR 0                [0x248600]      @0x24a4ec
   (0x24de20) :    CHECK_LEX 0               [0x218d87]      @0x24a4ec
   (0x24de3c) :    FIND 1520                 [0x2493cc]      @0x24a4ec
   (0x24de04) :    R_TOPONLY 1523            [0x21b5d4]      @0x24a4ec
   (0x24dde8) :    R_CLASS 1524              [0x248df0]      @0x24a4ec
   (0x24ddcc) :    USE 1527                  [0x2493cc]      @0x24a4ec
   (0x24ddb0) :    TYPE_BUILD_REF 1528       [0x249678]      @0x24a4ec
   (0x24e134) :    TYPE_END_SET 0            [0x249678]      @0x24a4ec
   (0x24e150) :    TYPE_END 1530             [0x0]           @0x24a4ec
   (0x24e0fc) :    CHECK_LEX 0               [0x218d87]      @0x24a4ec
   (0x24e118) :    FIND 1531                 [0x2493a4]      @0x24a4ec
   (0x24e0e0) :    BUILD 1534                [0x249b54]      @0x24a4ec
   (0x24e0c4) :    DEFINE 1536               [0x2493a4]      @0x24a4ec
   (0x24e0a8) :    EXPR_REF 1538             [0x20c9f0]      @0x24a4ec
   (0x24e08c) :    MODE_SET 1539             [0x1201]        @0x24a4ec
   (0x24e070) :    SYMBOL_SET_MODE 1541      [0x2493a4]      @0x24a4ec
   (0x24e054) :    VALUE 1542                [0x2493a4]      @0x24a4ec
   (0x24e038) :    SYMBOL_SET_TYPE_OF 1544   [0x249678]      @0x24a4ec
STRING (end)
   (0x24dfe4) :    CHECK_LEX 0               [0x0]           @0x24a484
   (0x24e000) :    FIND 1156                 [0x249304]      @0x24a484
   (0x24dfc8) :    EXPR_BUILD 1159           [0x20c910]      @0x24a484
   (0x24dfac) :    FLOW_BLOCK 1161           [0x2489dc]      @0x24a484
OUTPUT_EXPRESSION (end)
   (0x24e330) :    NOP 0                     [0x0]           @0x24a450
OUT_EXPR_S (end)
   (0x24e2dc) :    CHECK_LEX 0               [0x0]           @0x24a41c
   (0x24e2f8) :    FIND 1134                 [0x2492dc]      @0x24a41c
   (0x24e2c0) :    EXPR_BUILD 1137           [0x20ca28]      @0x24a41c
   (0x24e2a4) :    FLOW_BLOCK 1139           [0x2489a8]      @0x24a41c
   (0x24e26c) :    CHECK_LEX 0               [0x0]           @0x24a41c
   (0x24e288) :    FIND 1142                 [0x249e3c]      @0x24a41c
   (0x24e250) :    EXPR_BUILD 1145           [0x20c8d8]      @0x24a41c
   (0x24e234) :    FLOW_BLOCK 1147           [0x248974]      @0x24a41c
WRITELN (end)
   (0x24e1fc) :    FLOW_BLOCK 557            [0x248940]      @0x24a3e8
   (0x24e1c4) :    CHECK_LEX 0               [0x0]           @0x24a3e8
   (0x24e1e0) :    FIND 560                  [0x249e14]      @0x24a3e8
   (0x24e1a8) :    R_CURONLY 563             [0x249c40]      @0x24a3e8
   (0x24e18c) :    USE 564                   [0x249e14]      @0x24a3e8
   (0x24e170) :    FLOW_BLOCK 565            [0x24890c]      @0x24a3e8
   (0x248568) :    FLOW_BLOCK 568            [0x2488d8]      @0x24a3e8
   (0x24854c) :    SYMBOL_SET_FLOW 571       [0x249e14]      @0x24a3e8
   (0x248530) :    END 572                   [0x218984]      @0x24a3e8
   (0x248514) :    FLOW_FREE 573             [0x2488a4]      @0x24a3e8
WHILE (end)
```

Figure 5-6 continued

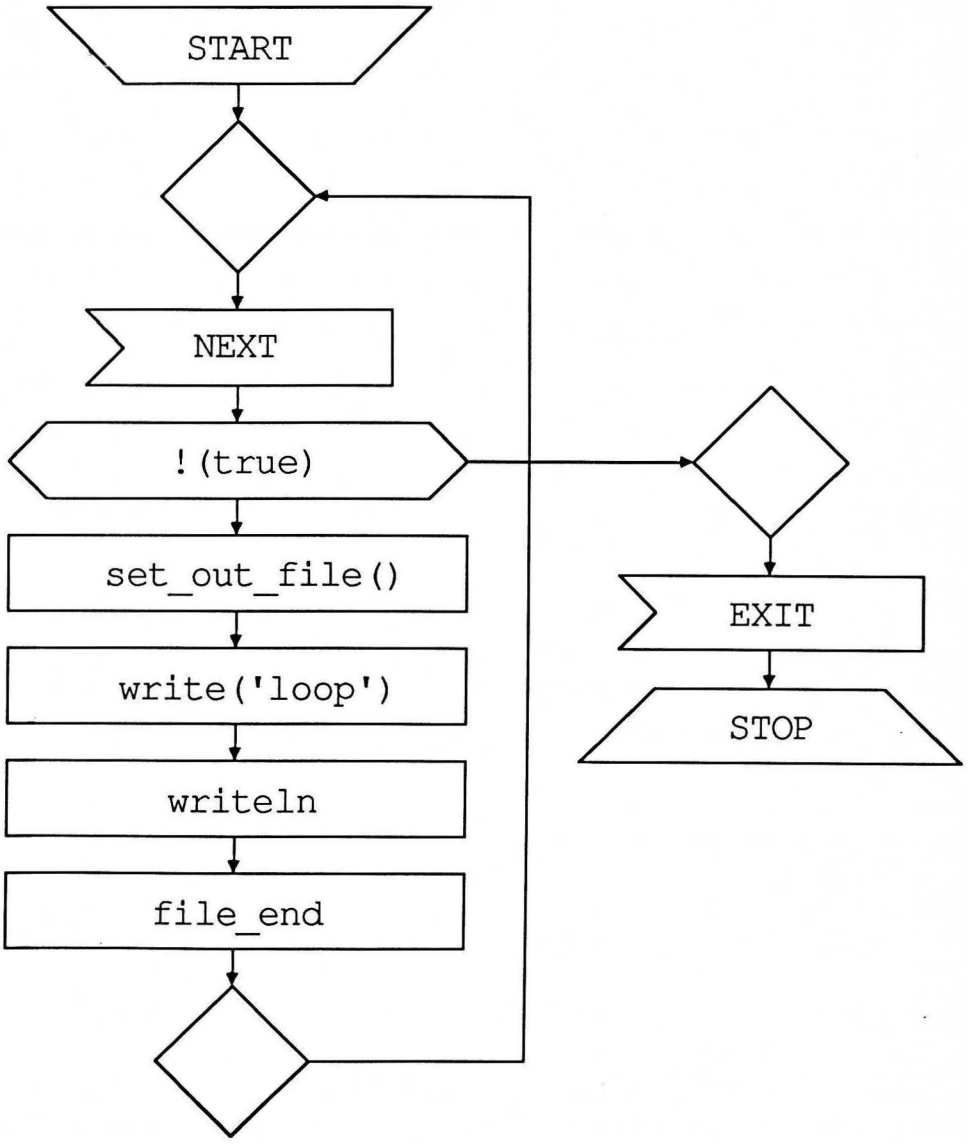

Figure 5-7: Flow Graph Representation of Example WHILE Statement

5.5. Incremental Compilation in PECAN

The system as described thus far would be appropriate for an α-type incremental compiler operating in the following manner. If the recompilable unit was taken to be a Pascal procedure then every time a node was changed in the AST, the SEMCOM_STMTs associated with the procedure in which the change was made could be unexecuted (effectively removing the interpretable code for that procedure) then the SEMCOM_STMTs representing the modified procedure could be executed to restore the flow graph.

However, PECAN is a β-type incremental compiler; it determines the smallest recompilable unit before incrementally compiling. The algorithm used by PECAN is described in §5.5.1.

5.5.1. General Algorithm

When a change is made to the AST, SEMCOM creates a list of SEMCOM_STMTs (the *new list*) corresponding to the new node. The list of SEMCOM_STMTs corresponding to the node as it was before the alteration is referred to as the *old list*. The old list and the new list are compared and the *area of difference* is established. The SEMCOM_STMTs preceding and following the area of difference in both lists are disregarded, in order to avoid unnecessary recompilation.

It is not sufficient to simply unexecute the resulting old list then execute the corresponding new list. It may well be that the area of difference represents only part of a construct. Its semantic validity may depend upon SEMCOM_STMTs representing the rest of the construct. For example, consider the Pascal statement

```
IF x = y THEN
  <statement>
ELSE
  <statement>;
```

If the identifiers x and y are declared as being of the same type then this will be a valid statement. If the identifier x is replaced by the identifier z then the validity of the condition depends upon the type of z. Clearly it is not enough to simply replace the flow graph code that determines the value of x with similar code for z. First, z must be checked for compatibility with y.

SEMCOM extends the new list to include SEMCOM_STMTs until all of the local effects of the change have been covered. The new list is unexecuted back to the point where the lists differed. The old list is then unexecuted, before the extended new list is executed. An update routine propagates changes throughout the rest of the program.

5.5.2. Implementation Details

A detailed description of how SEMCOM implements this algorithm requires an understanding of the workings of some of the lower-level SEMCOM functions.

The operation of the functions *head_merge*, *tail_merge*, *extend*, *remove* and *insert* will be described by reference to the diagrammatic representation of the old and new lists which appears in Figure 5-8. The old list is that list between the *oldp* and *oendp* pointers. The new list is that list between the *newp* and *nendp* pointers. The SEMCOM_STMTs with shaded bodies are those that form the area of difference.

5.5.2.1. *head_merge*

Figure 5-8(a) shows the state of the lists of SEMCOM_STMTs before the *head_merge* operation is performed. The old list is part of a longer list that represents the whole program - the *main list*. The new list exists separately. The *head_merge* operation moves the *oldp* and *newp* pointers down their respective lists until the SEMCOM_STMTs that they refer to are different. As the pointers are moved, the new list is merged into the old list, and the duplicate SEMCOM_STMTs are removed from the old list. Figure 5-8(b) shows the state of the lists after the *head_merge*.[10]

5.5.2.2. *tail_merge*

The *tail_merge* function is complementary to *head_merge*. Figure 5-8(b) shows the state of the lists before the *tail_merge* operation and Figure 5-8(c) shows the state afterwards. The duplicated SEMCOM_STMTs in the new list have been merged into the old list, and the corresponding old SEMCOM_STMTs have been discarded.

5.5.2.3. *extend*

The *extend* function moves the *nendp* pointer (effectively extending the new list) until it includes all of the SEMCOM_STMTs required to ensure that all of the local effects of the change are completed. As has been explained, the meaning of each construct in the language is given by semantic specification statements in terms of the current items. So the local effects of a change to the program will be reflected in those current items.

---

[10] SEMCOM_STMTs removed from the old list are shown in Figure 5-8(b) with no pointers pointing to them. In fact they are removed, one at a time, by *head_merge* yet they do not disappear from the diagrammatic representation until Figure 5-8(c). The discarded SEMCOM_STMTs appear in Figure 5-8(b) in order to make the operation of *head_merge* clear. The same is true of *tail_merge*, where the SEMCOM_STMTs that are removed do not disappear until Figure 5-8(d).

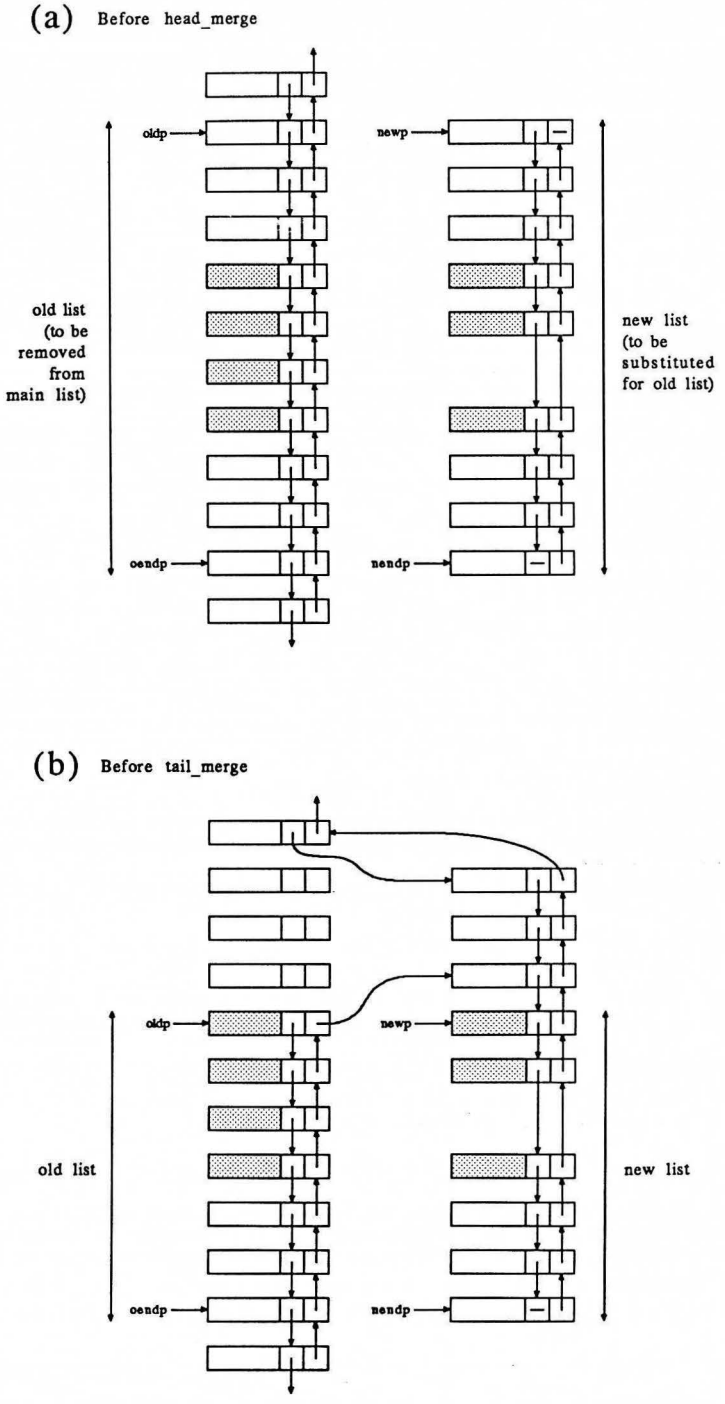

Figure 5-8: Effect of *head_merge*, *tail_merge* and *extend* upon the old and new lists

(continued next page)

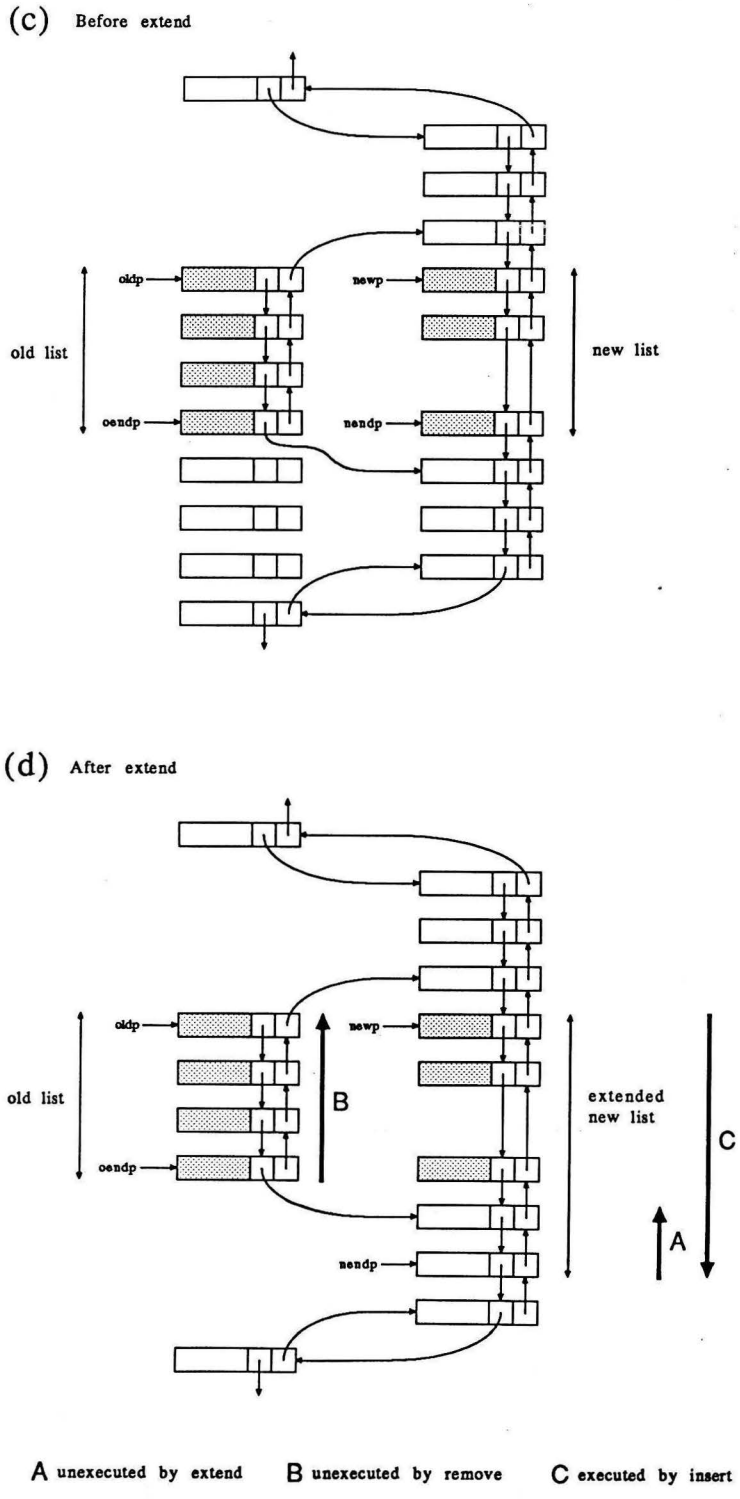

Figure 5-8 continued

Some of the semantic specification statements have a corresponding statement which must appear in order for the list of statements to provide a valid specification. For example a BEGIN statement (which marks the beginning of a new scope) must have an associated END; a PUSH statement must have an associated POP. These statements, which must follow certain other statements, will be referred to as *end bracket statements.* There are four end bracket statements (END, POP, TYPE and FLOW) which may be required by the occurrence of various *start bracket statements.*[11]

The *extend* function proceeds as follows. First the old list is scanned, in order to count the number of end bracket statements with no matching begin bracket statements in the old list. The new list is then scanned, and extended (if necessary) until

- it contains an unmatched end bracket statement corresponding to each such unmatched end bracket statement found in the old list; and
- each begin bracket statement in the new list has a matching end bracket statement.

Figure 5-8(c) shows the state of the lists before the *extend* operation, and Figure 5-8(d) shows their state afterwards. Because the new list has been merged into the old list (by *head_merge* and *tail_merge*, the only limit on how far *extend* can move the *nendp* pointer is the end of the complete list of SEMCOM_STMTs (*i.e.* the end of the program).

The *extend* function also performs the unexecution of the extended part of the new list (marked **A** in Figure 5-8(d)).[12]

5.5.2.4. *remove* and *insert*

The *remove* function unexecutes (in reverse order) each of the SEMCOM_STMTs in the old list (marked **B** in Figure 5-8(d)). Each SEMCOM_STMT is removed from the list after unexecution.

The *insert* function executes (in order) each of the SEMCOM_STMTs in the extended new list (marked **C** in Figure 5-8(d)).

---

[11] Not all of the start bracket statements for TYPE and FLOW have been identified.

[12] Inexplicably, Reiss makes no mention of this step in his description of PECAN's incremental compilation [Reiss 84a]. If this step is not taken, the extended part of the new list will soon be executed (by *insert*) without first having been unexecuted.

5.5.2.5. The Current Items and Execution and Unexecution

The functions *_SEMCOM_execute* and *_SEMCOM_unexecute* perform execution and unexecution respectively. Before a SEMCOM_STMT can be executed or unexecuted it has to be put into context; the values of the current items must be established. Before the *insert* function calls the *_SEMCOM_execute* function for the first time, it calls the *_SEMCOM_set_current* function to set the current items to the values that they should hold before the first SEMCOM_STMT in the new list. *_SEMCOM_set_current* moves backwards through the list of SEMCOM_STMTs preceding the new list, retrieving the values that were most recently assigned to each of the current items. Once values for all of the current items have been retrieved, execution can commence. Each time a SEMCOM_STMT is executed, the current items are updated accordingly.

Unexecution is handled slightly differently. Every time the *_SEMCOM_unexecute* function is called (by *extend* or *remove*) in order to unexecute a single SEMCOM_STMT, the values of the current items are determined. However, the *_SEMCOM_unexecute* function only determines the values of those current items which are relied upon in the unexecution of the SEMCOM_STMT in question.

5.5.2.6. Updating the Semantics

SEMCOM has four semantic support modules: the symbol table support module, the type support module, the expression support module, and the flow graph support module. When a SEMCOM_STMT is executed or unexecuted, two stages of processing are triggered:

- the flow graph representation is modified (as explained in §5.4); and
- information is passed to the relevant support module for processing after the execution and unexecution of all the SEMCOM_STMTs is completed.

In the second case, information is queued to a support module which adds that information to a list of operations it must perform when the execution and unexecution is finished.

When a definition of a name is created, modified or removed, all of the references to that name are queued with the symbol table support module for later checking. When a type reference cannot be immediately resolved (*i.e.* it relies upon a name in the symbol table) then that type is queued with the type support module. When an expression is modified it is queued with the expression support module for later resolution. When a flow graph operation is required, but cannot be performed in the first phase (*i.e.* it relies upon a name in the symbol table), that operation is queued with the flow graph support module.

When all of the execution and unexecution has been performed, the *SEMCOMupdate* function is invoked. That function calls each of the support modules in turn, requesting that they process the requests stored in their respective queues. Each call causes the support module in question to continue resolving items from its list until the list is empty. The dependencies between the modules are such that running down the list of one module can result in other requests being queued in any other support module except the symbol table support module. For that reason, the symbol table support module is forced to update first, then the other three support modules are called repeatedly until all of the lists are empty, at which point all of the effects of the original change have been propagated throughout the program.

5.5.2.7. Driving Routines - The Outer Level of SEMCOM

The functions described above (§5.5.2.1 to §5.5.2.4) are invoked by the externally visible (outer level) SEMCOM routines.

When SEMCOM is initialized it registers its interest in an event called ASPEN_$NODE_CHANGE. This event is triggered by the ASPEN module when a node in the AST is changed or deleted. The ASPEN_$NODE_CHANGE event passes, as a parameter, a pointer to the modified node in the AST. The event causes a call to the *sem_event_node* function, which determines whether the new node has been modified or deleted and calls *_SEMCOM_replace_list* or *_SEMCOM_remove_list* accordingly.

_SEMCOM_replace_list uses the *ASPENinq_semantics* function to find the head and tail of the list of SEMCOM_STMTs associated with the new node. Although the node has been changed, its associated SEMCOM_STMTs are still those of the old node (*i.e.* the old list).

A new list of SEMCOM_STMTs is generated, representing the semantics of the new node. The pointers to the head and tail of this list (the new list) are stored in the AST, overwriting the AST's pointers to the old list. After the *head_merge* and *tail_merge* functions merge the new list into the main list, the main list of SEMCOM_STMTs accurately reflects the semantics of the program represented by the AST.

Incremental compilation may now begin. The values of the *oldp*, *oendp*, *newp* and *nendp* pointers are known. These values are used to call *head_merge*, *tail_merge*, *extend*, *remove* then *insert*.

_SEMCOM_remove_list performs similar tasks to those performed by *_SEMCOM_replace_list*. However, there is no new list, so the five functions are called with null values for the *newp* and *nendp* pointers. Effectively, *remove* is the only function of these five which will do anything when called by *_SEMCOM_remove_list*.

The *SEMCOMupdate* function is invoked from the main loop in the outermost level of PECAN (*pascalmain.c*), to update the semantics after the execution and unexecution is completed.

Chapter 6
Modifications to PECAN

6.1. Aim of the Modifications

The aim of this thesis project is to find some way of comparing the PECAN approach to incremental compilation (β-type incremental compilation) with an α-type incremental compilation method. As mentioned in §2.2.2, balancing the costs of α-type and β-type incremental compilation is the fundamental design question in the area of incremental compilation.

To this end, the SEMCOM module has been modified so that PECAN can support three different types of incremental compilation:

- incremental compilation (β-type) as before;
- procedure compilation (α-type incremental compilation with the smallest enclosing Pascal procedure or function[1] or main program as its recompilable unit); and
- complete compilation (α-type incremental compilation with the entire program as its recompilable unit).

Further, the programmer is given the ability to specify that recompilation should be performed *automatically* (as before) or *manually* (*i.e.* at the programmer's request).[2]

Procedure compilation will occur automatically (regardless of whether compilation is *automatic* or *manual*) if

- the programmer makes an editing change to a node in the AST which is not enclosed by the same procedure as was the last node to be changed (*i.e* the programmer has moved out of a procedure); and
- the procedure which encloses the last node which was changed has not already been recompiled.

---

[1] Throughout this chapter the word "procedure" will be used to refer to a procedure *or* function (except where the context indicates otherwise).

[2] For a discussion of the question of when to trigger recompilation, see §2.3.2.2.

The main interest of this project is with procedure compilation; complete compilation was included for curiosity.

Effectively, these modifications enable a comparison to be made of the relative merits of the approach to incremental compilation taken by PECAN and that taken by the Magpie system (see §3.3.5). Magpie performs its recompilation on a procedure basis. When the programmer has finished making editing changes to a procedure, that procedure is recompiled in the background. It is not practicable to implement background compilation in PECAN. Nevertheless the two methods can be compared within the PECAN system. By setting compilation to *manual*, and allowing PECAN to recompile each procedure after a number of editing changes have been carried out within that procedure, PECAN can be made to approximate the Magpie approach.

6.2. Generality of the Modifications

It will be recalled that, since page 19, PECAN has been considered not as an environment generator but as a Pascal environment. However, when modifying the SEMCOM module, thought must be given to that module's generality and whether any of the modifications are language specific. There is one modification that has been made to SEMCOM as part of this thesis project which assumes that the supported language is Pascal. One step in procedure compilation involves finding a node's enclosing procedure in the AST.[3] This is performed by moving up through the tree until a BLOCK node is found. BLOCK nodes are defined in the specification for Pascal, but there is no good reason to suppose that the specification for any other language will define its recompilable unit as a BLOCK.[4]

This flouting of generality can be justified for the purposes of this experimental comparison of compilation methods. If these modifications to PECAN were to be implemented in a more concrete fashion, the language specification could be altered to allow an explicit statement that a given construct is a recompilable unit. Provision for tagging constructs already exists.[5] Given the fact that the modifications made as part of this project were intended only to compare two different approaches to incremental compilation, it was deemed unnecessary to alter the definition of the specification language.

---

[3] As described in §6.4.3.

[4] Indeed, another language may specify more than one recompilable unit. In Pascal, BLOCK is sufficient as it makes up part of all three recompilable units: procedures, functions and the main program.

[5] *e.g.* the COMMENT label, used to indicate that the construct can be followed by a comment.

6.3. Ideal Modifications

When an entire procedure is recompiled after a number of modifications have been made, the compiler has to replace the flow graph representation of the old procedure with a flow graph representation of the new procedure. Parts of a flow graph representation are removed when SEMCOM_STMTs are unexecuted. However, in the case of procedure compilation, it would be useful if the flow graph representation of the procedure could be removed in one step before a new representation is constructed by executing SEMCOM_STMTs. Unfortunately, the module which maintains the flow graph representation (the FLOW module) does not provide a function to remove large sections of the flow graph representation in one operation. It was decided to limit the modifications made in this thesis project to one module of the PECAN system (the SEMCOM module). Accordingly, no change has been made to the FLOW module. Removal of the flow graph representation of a procedure is implemented using the *_SEMCOM_unexecute* function.[6]

When comparing the results of a number of tests (see §6.6), the cost of unexecuting SEMCOM_STMTs in order to remove the flow graph representation of a procedure is ignored on the basis that it would be possible to perform the same operation in one step.

[6]The same is true when removing the flow graph representation of an entire program during complete recompilation.

6.4. Actual Implementation Details

Details of the SEMCOM module code that has been modified or added in the course of this thesis project are given in Appendix B.

6.4.1. The Compilation Monitor

The SEMCOM module has been modified so as to provide compilation information in a window (the compilation monitor). For incremental compilation (as previously implemented) the compilation monitor displays:

- the number of SEMCOM_STMTs eliminated by *head_merge*;
- the number of SEMCOM_STMTs eliminated by *tail_merge*;
- the number of SEMCOM_STMTs by which *extend* extends the new list;
- the number of SEMCOM_STMTs unexecuted and removed by *remove*; and
- the number of SEMCOM_STMTs executed by *insert*.

This new window allows the programmer to set the type of compilation (*incremental*, *procedure* or *complete*) and to toggle the *automatic/manual* switch. There is also a COMPILE button which forces SEMCOM to compile using whichever compilation method was last chosen.[7] Using the COMPILE button has no effect if the compilation is set to *incremental* for the very good reason that incremental compilation is meaningless unless there is an amended node from which to construct a new list.

The compilation information, together with information about which compilation method is current, is displayed in the compilation monitor. When this information can no longer be displayed on the screen, the screen scrolls to keep up with the latest information. The rest of the new window's commands concern moving around within the window.

It should be noted that it is possible to do some fairly horrible things to the SEMCOM representation of the AST by using the SEMCOM window in a naive way. For example, if the user were to set compilation to *incremental* and *manual* then no change to the AST would result in any compilation being performed. Even if compilation were then set to *automatic*, the effect of the changes made

---

[7] When first invoked, the modified SEMCOM module is ready to perform *automatic incremental* compilation, just as it would have done before it was modified.

while compilation was set to *manual* would not be reflected in the SEMCOM representation of the program's semantics. The modifications to SEMCOM have been made for experimental purposes only. Although they provide a fairly robust view, that view is not intended to be foolproof.

6.4.2. Incremental Compilation

Incremental compilation is performed in precisely the same way as before except that calls to the various lower level functions have been moved into different functions.

6.4.3. Procedure Compilation

In order to perform incremental compilation on a procedural basis, SEMCOM makes a copy of the list of SEMCOM_STMTs which are associated with the procedure that is being edited before changes are made to that procedure. When a change is made to the AST, a list of SEMCOM_STMTs corresponding to the changed node is created and merged into the main list at the appropriate place. When compilation is triggered[8] the SEMCOM_STMTs in the list that represents the old procedure are unexecuted,[9] then the corresponding SEMCOM_STMTs in the main list are executed.[10]

The effect of this is much the same as if the entire procedure had been modified then incrementally compiled in the usual PECAN fashion, except that

- there is no attempt to find the area of difference (*i.e.* no use of *head_merge* or *tail_merge*); and
- there is no extension of either list (*i.e.* no use of *extend*).

The *extend* function is not required because procedural compilation is recompiling a recompilable unit. A recompilable unit has been defined[11] as a construct of the language such that no change to that construct can affect the meaning of any part

---

[8] Procedure compilation can be triggered in one of three ways: manually (by use of the COMPILE button), automatically (every time a change is made), or because the programmer has edited a node of the AST that is outside the procedure.

[9] For a discussion of the reasons why these SEMCOM_STMTs are unexecuted, see §6.3.

[10] Execution commences after the current items have been restored to their appropriate values in the manner described in §5.5.2.5.

[11] See page 4.

of the program outside that construct. In other words, no change within a procedure can cause any local effects in the semantic specification statements beyond the end of that procedure; the use of the *extend* function would not result in any extension of the new list.

6.4.4. Complete Compilation

The modified SEMCOM module performs complete compilation in the following manner. When a change is made to the AST, a list of SEMCOM_STMTs corresponding to the changed node is created and merged into the main list, then the *remove* function is applied to the old list in order to remove the corresponding nodes from the flow graph representation of the program. When compilation is triggered[12] the SEMCOM_STMTs in the main list are unexecuted, then executed.

This approach could be (uncharitably) described as being a bit "quick and dirty". After all, unexecuting the main list involves unexecuting SEMCOM_STMTs which have not yet been executed (specifically, all of the SEMCOM_STMTs that have been merged into the main list after changes to the AST). The *_SEMCOM_unexecute* function is sufficiently robust to handle this without incident, because it does not attempt to remove any non-existent nodes from the flow graph representation.

6.5. Drawing Comparisons

6.5.1. Choosing an Appropriate Benchmark

Three possible benchmarks were considered for comparing the efficiency of the different methods of incremental compilation implemented by the modified SEMCOM module: *elapsed time*, *code complexity* and *counting SEMCOM_STMTs*.

6.5.1.1. Elapsed Time

The main problem with measuring elapsed time is that it is affected in unpredictable ways by such diverse and uncontrollable factors as the number of users on the machine, the amount of free memory available, *etc.* There is no way to predict whether a particular method will be benefited by the idiosyncracies of the system on which the tests are carried out (or the state of the machine at the time at which the tests are carried out). This method is plainly unacceptable.

---

[12] Complete compilation can be triggered in either of two ways: manually (by use of the COMPILE button), or automatically (every time a change is made).

6.5.1.2. Code Complexity

Profiling the C code that is actually executed by PECAN (*i.e.* counting C statements) would provide the most detailed possible comparison of compilation methods. This approach assumes that all of the functions which are invoked by the various compilation methods are provided by code which is roughly equivalent in its efficiency. Otherwise, one compilation method could compare unfavourably with another for no other reason than that it made frequent use of a function which was inefficiently written. This approach was deemed too dependent upon the implementation of PECAN to be a good benchmark.

6.5.1.3. Counting SEMCOM_STMTs

Another approach is to count the SEMCOM_STMTs that are processed. Rather than comparing the PECAN code executed by each method (as done when comparing code complexity) this approach examines the amount of the program under development that each method recompiles. No assumptions need be made about the relative efficiency of PECAN functions.

For each compilation method, the compilation monitor provides information on the number of SEMCOM_STMTs that have been executed and unexecuted and (in the case of incremental compilation) the number of SEMCOM_STMTs that have been eliminated by *head_merge* and *tail_merge*, and the extent to which the new list has been extended. From this information it is possible to derive a single number of SEMCOM_STMTs for comparison purposes. This number will be referred to as Δ. For incremental compilation, the number (Δ_I) is

- the number of SEMCOM_STMTs unexecuted by *extend*; *plus*
- the number of SEMCOM_STMTs unexecuted by *remove*; *plus*
- the number of SEMCOM_STMTs executed by *insert*.

For both procedure and complete compilation, the number (Δ_P or Δ_C) is

- the number of SEMCOM_STMTs executed.

Note that, for procedure compilation and complete compilation, the number of SEMCOM_STMTs unexecuted is ignored (see §6.3). Counting SEMCOM_STMTs is the preferred method of comparison.

6.5.2. A Cautionary Note

Before comparing Δ-values for the test cases, it is important to consider some inadequacies in the chosen approach to comparing compilation methods. The approach is deficient in three ways:

- Procedure compilation and complete compilation have been built upon a system which was designed specifically for incremental compilation. PECAN's method of incremental compilation is being compared with that of Magpie (and traditional complete recompilation) within a framework which was constructed specifically for PECAN's method. Therefore, it must be expected (in a general sense) that the implementations of procedure and complete compilation will not be the most efficient.

- Counting SEMCOM_STMTs makes no allowance for the considerable computation required to perform semantic updating after execution and unexecution (as described in §5.5.2.6). Comparing Δ-values in the suggested manner assumes that the amount of computation required by the updating process is proportional to the number of SEMCOM_STMTs executed and unexecuted. This assumption would appear to be reasonable; no one compilation method could be expected to require more updating per SEMCOM_STMTs than any other. However, this assumption has not been properly validated.

- Counting SEMCOM_STMTs takes no account of the computation performed by the *extend* function in determining how far to extend the new list. This difficulty can be obviated by assuming that the computational cost of extending the new list by one SEMCOM_STMT is negligible when compared with the cost of executing or unexecuting one SEMCOM_STMT. This assumption is not necessarily invalid, but is by no means safe.

  A further extension of this thesis project would have been

  - to prove this assumption; or

  - to develop a method of incorporating the cost of extending the new list into the comparison method.

These drawbacks must be considered when evaluating the results of tests described in §6.6.

6.6. Testing

To compare the different methods of compilation, a suite of Pascal programs was prepared. These programs were modified in various ways and Δ-values were calculated for each of the compilation methods.

6.6.1. Choosing Test Programs

When preparing the suite of test programs, a major factor constraining the choice of program was PECAN itself. PECAN will only support the development of small programs.[13] Given that the test programs were restricted in size it was decided to use examples which were typical of the programs written by programmers when learning to code in Pascal. Four programs were used: two from [Findlay 81], one from [Jensen 78], and one from the author's salad days. These programs are listed in Appendix C.

6.6.2. Modifications

It is important that the modifications made to the test programs reflect the sorts of changes that programmers are likely to make to Pascal code during program development. Unfortunately, literature on this topic proved undiscoverable.[14]

Any consideration of the manner in which programmers modify programs is complicated by the fact that the environment in which the program is being developed may effect the way in which programs are debugged. For example, if the environment recompiles small changes immediately and quickly then the programmer may be encouraged to move freely around the source code when debugging. However, if the environment pauses to recompile each procedure after editing changes have been made within that procedure then the programmer may be tempted to stay within that procedure until *all* of the intended changes have been made.

The sorts of editing changes made during program development are strongly linked to the errors that programmers tend to generate. After all, a major part of the debugging process is the removal of syntactic and semantic errors from the source code. The authors of [Garlan 84] claim that four errors account for 90% of all compiler error messages for Pascal programs developed by novice programmers.

---

[13] One Pascal program of a mere 150 lines proved too large.

[14] Methods of measuring a programmer's aptitude for debugging are discussed in [Weinberg 71] (see pages 174-175). Unfortunately, no mention is made of the sorts of editing changes that apt, or inapt, programmers make when debugging.

In order of frequency these are:

1. variable not declared;
2. variable declared, but not used;
3. variable declared and used, but not initialized; and
4. type mismatch

(of these four, only the first and the fourth are recognised as errors by PECAN).

Armed with this information (and the author's well-developed intuitions regarding the sorts of editing changes made during the development of a Pascal program) a series of tests were designed. These tests are intended to be indicative of the kinds of changes which programmers make.

Where a test required an initially incorrect program, the correct program was modified so that it was incorrect before modifications were performed in order to return the program to its original state. Eight tests were carried out.

1. *test1.p* (§C.1)
   4 occurrences of the same (undefined) variable were changed to a defined variable (*scalarproduct*). All of the occurrences were in the same procedure (*multiplymatrices*).
   $\Delta_I = 436 \quad \Delta_P = 640 \quad \Delta_C = 2018$

2. *test1.p* (§C.1)
   All 10 occurrences of the constant n were replaced with the integer constant 10. The constant n occurred in all 3 procedures. The changes were made in the order in which the instances of n occurred.
   $\Delta_I = 1640 \quad \Delta_P = 2811 \quad \Delta_C = 2128$

3. *test2.p* (§C.2)
   A single change was made to the definition of the constant *pi*.
   $\Delta_I = 60 \quad \Delta_P = 904 \quad \Delta_C = 925$

4. *test2.p* (§C.2)
   4 occurrences of the same (undefined) variable were changed to a defined variable (*degrees*). All 4 occurrence were in the main program.
   $\Delta_I = 428 \quad \Delta_P = 904 \quad \Delta_C = 925$

5. *test2.p* (§C.2)

   The invocation of the *tan* function was replaced by an expression which produced the same result,[15] then the *tan* function was removed from the program.

   $\Delta_I = 472 \quad \Delta_P = 1652 \quad \Delta_C = 808$

6. *test3.p* (§C.3)

   A single corrective change was made to a misspelt function call in the main program.

   $\Delta_I = 100 \quad \Delta_P = 147 \quad \Delta_C = 447$

7. *test3.p* (§C.3)

   All 3 occurrences of an undefined identifier within the *factorial* function were changed to references to that function.

   $\Delta_I = 436 \quad \Delta_P = 147 \quad \Delta_C = 447$

8. *test4.p* (§C.4)

   5 more calls to the *try* function were added to the main program.

   $\Delta_I = 285 \quad \Delta_P = 649 \quad \Delta_C = 670$

Full details of all of the compilation information extracted for each of these tests are given in Figure 6-1. In the case of incremental compilation, the column headings are

H - SEMCOM_STMTs disposed of by *head_merge*;
T - SEMCOM_STMTs disposed of by *tail_merge*;
E - SEMCOM_STMTs by which *extend* extends the new list;
R - SEMCOM_STMTs removed and unexecuted by *remove*; and
I - SEMCOM_STMTs inserted and executed by *insert*.

In the case of procedure compilation and complete compilation the column headings are

UN - SEMCOM_STMTs unexecuted; and
EX - SEMCOM_STMTs executed.

[15] *i.e. tan(degrees*pi/180)* was replaced by *sin(degrees*pi/180)/cos(degrees*pi/180)*.

6.6.3. Comparison of Results

In this section, the results are interpreted by simple comparison of Δ-values. The questions raised (in §6.5.2) about the efficacy of comparing Δ-values are ignored for the moment.

In 5 out of the 8 tests[16] incremental compilation performed better than procedure compilation which performed better than complete compilation.[17] In 2 of the tests[18] procedure compilation did not perform as well as complete compilation due to the large number of procedures which were edited.

Only in test 7 was incremental compilation not the most efficient of the compilation methods (although it still performed better than complete compilation). In that test, 3 changes were made within a function. That function is so short that it can easily be understood how 3 changes required more work to compile separately than did the whole function.

On the basis of these results, it would seem that unless the changes made within a recompilable unit affect a substantial amount of that recompilable unit (*i.e.* either the unit is very small, or the number of changes is large) then incremental compilation is more efficient than procedure compilation.

In other words (and making no allowance for the computational cost of extending the new list) β-type incremental compilation is more efficient than α-type incremental compilation.

It is also interesting to note that the *head_merge* and *tail_merge* functions discard very few SEMCOM_STMTs. This raises doubts as to the need to reduce the old list and the new list to the area of difference, when incrementally compiling Pascal structures.

---

[16] Tests 1, 3, 4, 6 and 8.

[17] *i.e.* $\Delta_I < \Delta_P < \Delta_C$.

[18] Tests 2 and 5.

test1.p 2018 SEMCOM_STMTS

(1) 4 undefined variables changed

Incremental	H	T	E	R	I	
	0	1	27	1	28	
	0	1	88	1	89	
	0	1	80	1	81	
	0	1	19	1	20	
total	0	4	214	4	218	Δ_I = **436**

Procedure	UN	EX	
	640	640	Δ_P = **640**

Complete	UN	EX	
	2018	2018	Δ_C = **2018**

Figure 6-1: Results of Modifying Test Programs

(continued next page)

test1.p 2018 SEMCOM_STMTS

(2) Change all 10 *n*s to 10

Incremental	H	T	E	R	I	
	1	1	29	7	31	
	0	0	29	2	47	
	0	0	12	7	14	
	0	0	12	2	30	
	0	0	12	7	14	
	0	0	125	2	143	
	1	1	29	7	31	
	0	0	29	2	47	
	0	0	12	7	14	
	0	0	12	2	30	
	0	0	12	7	14	
	0	0	12	2	30	
	1	1	37	7	39	
	0	0	37	2	55	
	0	0	12	7	14	
	0	0	12	2	30	
	0	0	12	7	14	
	0	0	12	2	30	
	0	0	12	7	14	
	0	0	216	2	234	
total	3	3	675	90	875	Δ_I = **1640**

Procedure	UN	EX	
	1991	1991	
	370	370	
	450	450	
total	2811	2811	Δ_P = **2811**

Complete	UN	EX	
	2128	2128	Δ_C = **2128**

Figure 6-1 continued

(continued next page)

test2.p 925 SEMCOM_STMTS

(3) Single change to value of constant at outer level

Incremental	H	T	E	R	I	
	1	8	21	9	30	Δ_I = **60**
Procedure	UN	EX				
	904	904				Δ_P = **904**
Complete	UN	EX				
	925	925				Δ_C = **925**

(4) 4 undefined variables changed

Incremental	H	T	E	R	I	
	0	1	38	1	39	
	0	1	34	1	35	
	0	1	52	1	53	
	0	1	86	1	87	
total	0	4	210	4	214	Δ_I = **428**
Procedure	UN	EX				
	904	904				Δ_P = **904**
Complete	UN	EX				
	925	925				Δ_C = **925**

(5) Replace call to *tan*(X) with *sin*(X)/*cos*(X), then delete *tan* function

Incremental	H	T	E	R	I	
	1	1	21	82	30	
	0	0	21	0	30	
	0	0	5	2	121	
	0	0	0	159	1	
total	1	1	47	243	182	Δ_I = **472**
Procedure	UN	EX				
	904	945				
	945	707				
total	1849	1652				Δ_P = **1652**
Complete	UN	EX				
	808	808				Δ_C = **808**

Figure 6-1 continued

(continued next page)

test3.p 447 SEMCOM_STMTS

(6) Single undefined function call changed

Incremental	H	T	E	R	I	
	0	1	49	1	50	Δ_I = **100**
Procedure	UN	EX				
	147	147				Δ_P = **147**
Complete	UN	EX				
	447	447				Δ_C = **447**

(7) 3 occurrences of undefined function identifier changed

Incremental	H	T	E	R	I	
	0	1	99	1	100	
	0	1	68	1	67	
	0	1	49	1	50	
total	0	3	216	3	217	Δ_I = **436**
Procedure	UN	EX				
	147	147				Δ_P = **147**
Complete	UN	EX				
	447	447				Δ_C = **447**

test4.p 670 SEMCOM_STMTS

(8) 5 more calls to *try* added to main body

Incremental	H	T	E	R	I	
	0	0	0	1	56	
	0	0	0	1	56	
	0	0	0	1	56	
	0	0	0	1	56	
	0	0	0	1	56	
total	0	0	0	5	280	Δ_I = **285**
Procedure	UN	EX				
	649	649				Δ_P = **649**
Complete	UN	EX				
	670	670				Δ_C = **670**

Figure 6-1 continued

Chapter 7
Conclusions

As stated in §6.6.3, the test results suggest that β-type incremental compilation (where the smallest amount of recompilation is performed after each editing change) is more efficient than α-type incremental compilation (where a structure of the programming language is chosen as the recompilable unit). However, there are a number of deficiencies in the comparison method chosen (as explained in §6.5.2). Of these deficiencies, the one which favours incremental compilation the most is the third: the fact that no account was taken of the computation performed by the *extend* function in order to determine how far to extend the new list. In the tests described in §6.6, some 35% of all of the SEMCOM_STMTs executed and unexecuted during incremental compilation were unexecuted by the *extend* function (*i.e.* the new list was extended to include those SEMCOM_STMTs). This indicates that the cost of extending the new list significantly affects the total cost of incremental compilation in PECAN.

A more comprehensive comparison of the compilation methods would have taken account of the cost of the *extend* function. Profiling the C code which is actually executed by PECAN in each case would provide such a comparison. That method was not adopted for this project because it is too dependent upon the implementation of PECAN (see §6.5.1.2). However, profiling the code would be an appropriate benchmark if the environment in which the comparisons were made was not biased towards one method of compilation, as PECAN was towards incremental compilation.

The comparisons that have been made between α-type and β-type incremental compilation do not allow any plenary statements to be made about the relative efficiency of the two methods. However, the performance of incremental compilation is not spectacularly better than that of procedure compilation, especially when the bias of the comparison method towards incremental compilation is taken into account. The results suggest that the gains in efficiency associated with β-type incremental compilation are so small that they do not justify the large amount of programming work, and structural overheads, required to implement such a compilation mechanism. β-type incremental compilation is faster, but not significantly faster, than α-type incremental compilation.

PECAN is a useful tool with which to test and demonstrate various aspects of programming environment design. However, it is only of limited use for examining general aspects of incremental compiler design. The entire structure of PECAN, from its language specification to the internal representation of its compilation module, is oriented towards β-type incremental compilation. The experiments carried out as part of this thesis project demonstrate that PECAN is not the ideal environment in which to compare various methods of incremental compilation.

The other achievements of this project are the thorough description of PECAN's compilation mechanism, and the implementation of the semantic actions view (a robust and useful view into the PECAN system).

Appendix A
The Semantic Actions View

A.1. The View and its Functions

A new view has been developed for the PECAN system. This view provides a list of the SEMCOM_STMTs associated with the current node, as highlighted in the SDE and other program views. Information about the type of the current node and its position in the tree is also provided. Buttons are provided which allow the window to be scrolled so that all of the list may be examined. Other buttons provide tree traversal commands. The view will respond to changes of the current node in other views, and will cause changes to be reflected in other views when the tree traversal commands are used.

An example PECAN screen, showing the semantic actions view, is reproduced in Figure A-1. The SDE and the flow view are on the left side of the screen, the semantic actions view is on the right. The SDE's cursor indicates the *factorial* identifier, and the flow view's cursor indicates the statement which assigns a value to that identifier.

The semantic actions view indicates that the current AST node is an IDENTIFIER node. It is the first of two children, and has one child of its own. The list of SEMCOM_STMTs that follows is that list associated with the parent of the current node. Those SEMCOM_STMTs associated with the current node are indicated by arrows ("->") and are separated from the surrounding SEMCOM_STMTs by two horizontal lines. The SEMCOM_STMTs associated with the parent of the current node are displayed in order to put the current node's SEMCOM_STMTs into context.

SEMCOM_STMTs are displayed in the following form:[1]

(location) : name index [value] @ pointer into AST

The index is displayed as a decimal number. All other numbers are hexadecimal.

---

[1] In the same form as they are displayed by the *_SEMCOM_dump* function in *semcommain.c*.

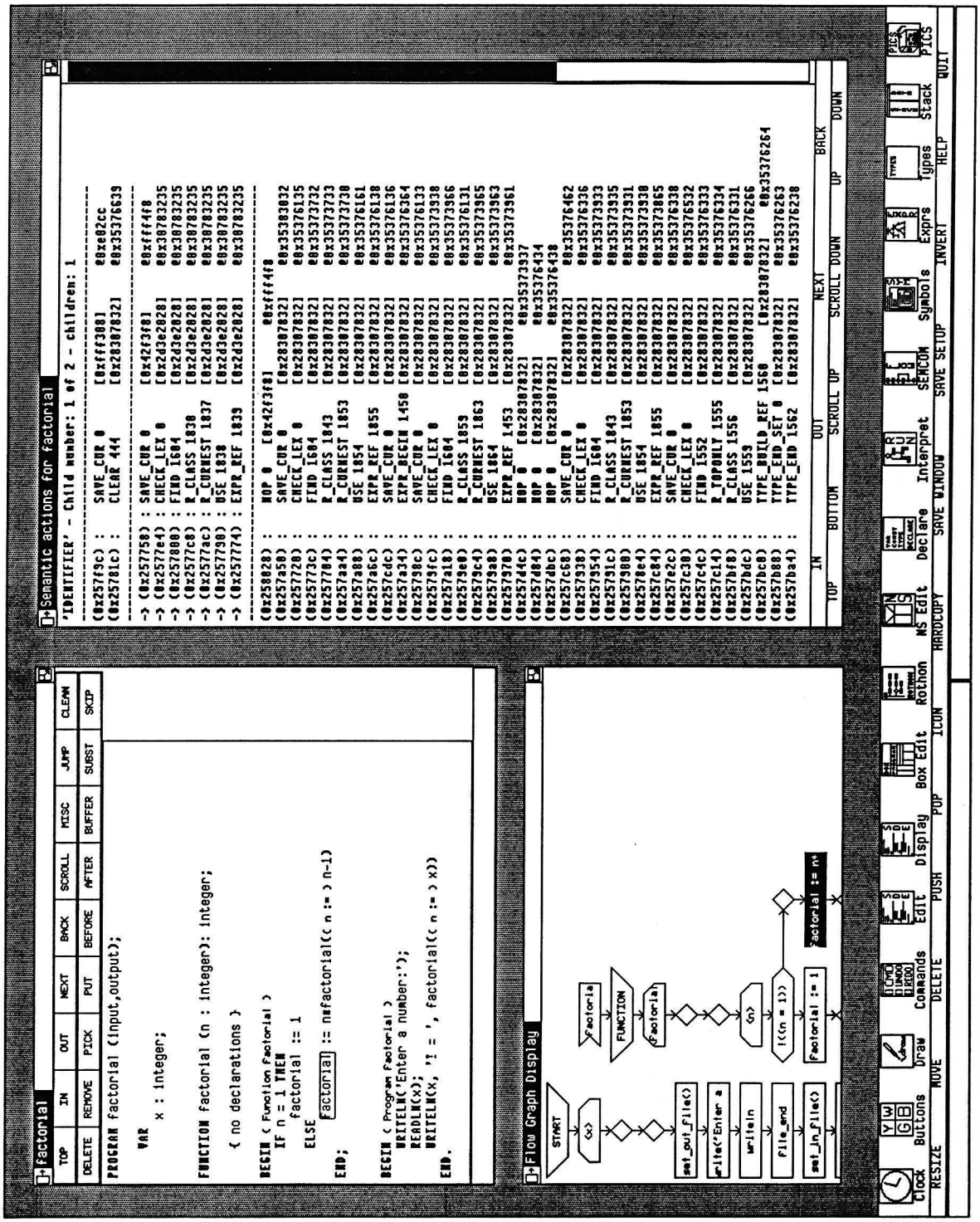

Figure A-1: The Semantic Actions View

The scroll bar on the right side of the view indicates that approximately two thirds of the whole list[2] is currently displayed. The window onto the list can be scrolled to a desired point in the list by using the *mouse* to *click* on the corresponding point on the scroll bar, or by using the scroll buttons (TOP, BOTTOM, SCROLL UP, SCROLL DOWN, UP and DOWN).[3]

The tree traversal buttons move the current node around the AST.[4] IN moves to the first (leftmost) child of the current node, and OUT moves to its parent. NEXT moves to the current node's next sibling, and BACK moves to its previous sibling. The view is updated after each tree traversal command, and an event is triggered so that other views will also reflect the change.

A.2. Implementation Details

The semantic action view is implemented by a new module called SAWDUST.[5] The view is designed to be completely compatible with existing views. The event passing system (provided by the PLUM module, and described in §4.2.3) is used to provide a clean interface between SAWDUST and existing modules. The formatting, tracing and function-naming conventions adopted in other PECAN modules have been followed in SAWDUST.

---

[2] *i.e.* the list of SEMCOM_STMTs associated with the parent of the current node.

[3] UP moves the window up by one quarter of a screen; SCROLL UP moves the window up by a whole screen.

[4] More correctly, the tree traversal buttons affect which node of the AST is considered the current node.

[5] SAWDUST stands for Semantic Action Window Display Using Several Tiles. This is a somewhat contrived acronym, but it pales into insignificance when compared with some of the acronyms which are used to name PECAN modules.

Examples range from the utilitarian
ASH - A Screen Handler,
APIO - Apollo Input Only Package (an anagrammatical acronym),
MFE - MAPLE Front End, and
VDI - Virtual Device Interface
through the fairly plausible
SGP - Simple Graphics Package,
BRIM - Brown Image Format, and
PLUM - Programming Language Utility Module
rising to the giddy heights of
BALSA - Brown University Algorithm Simulator and Animator, and
WILLOW - Wonderful Integrated Language for Laying-Out Windows.
Regrettably, the meanings of MAPLE and TULIP are unknown.
In this context, SAWDUST seems almost credible as an acronym.

The SAWDUST module consists of four files:

sawdust.h (§A.3)
The external header file.
Lists the externally accessible SAWDUST functions and gives details of the module's trace facilities.[6]

sawdust_local.hi (§A.4)
The local header file.
Includes a definition of the SAWDUST_SEMCOM_STMT type, which is identical in structure to the SEMCOM_STMT type but is defined in this way because the SEMCOM_STMT type is not externally accessible.

sawdustmain.c (§A.5)
Defines the SAWDUST window (using the WILLOW module from the Brown Workstation Environment) and displays SEMCOM_STMTs (using the VT module which provides a virtual terminal). Window movement and re-sizing is handled by WILLOW.

sawdustbutton.c (§A.6)
Button handling routines.

---

[6]Note that PECAN's *main* function (contained in *pascalmain.c*) is modified so as to invoke the *SAWDUSTinit* function and to allow trace information to be passed to the *SAWDUSTtrace* function. The previously unused Z debug switch was utilized. Invoking PECAN with the -DZ*n* option will cause the number *n* to be passed to *SAWDUSTtrace*.

A.3. Program Listing: *sawdust.h*

```
/*****************************************************************************/
/*                                                                           */
/*              sawdust.h                                                    */
/*                                                                           */
/*      External definitions for the Semantic Actions Window                 */
/*                                                                           */
/*****************************************************************************/
/*      James Popple  August 1987                                            */


/*****************************************************************************/
/*                                                                           */
/*      Tracing definitions -- use "-DZn" switch, where n gives              */
/*                             the type(s) of trace                          */
/*                                                                           */
/*****************************************************************************/

#define SAWDUST_TRACE_OFF       0
#define SAWDUST_TRACE_ON        1           /* external calls       */
#define SAWDUST_TRACE_INT       2           /* internal calls       */
#define SAWDUST_TRACE_DEBUG     4           /* debug                */


/*****************************************************************************/
/*                                                                           */
/*      Routine definitions                                                  */
/*                                                                           */
/*****************************************************************************/

                                            /* sawdustmain.c        */
extern              SAWDUSTinit();
extern              SAWDUSTtrace();


/* end of SAWDUST.h */
```

A.4. Program Listing: *sawdust_local.hi*

```
/*************************************************************************/
/*                                                                       */
/*              sawdust_local.h         (from sawdust_local.hi)          */
/*                                                                       */
/*      Local definitions for Semantic Actions Window                    */
/*                                                                       */
/*************************************************************************/


#include <aspen.h>
#include <flow.h>
#include <ash.h>
#include <maple.h>
#include <vt.h>
#include <willow.h>
#include <symbols.h>
#include <type.h>
#include <expr.h>
#include <semcom.h>
#include <acer.h>
#include <sawdust.h>


/*************************************************************************/
/*                                                                       */
/*      Data structures                                                  */
/*                                                                       */
/*************************************************************************/


#ifndef SAWDUST_MAIN
#define PLUM_INCLUDE_ONLY
#endif


Mode SawdustDefs Is

Type ASPEN_NODE From Mode ASPEN External;

SAWDUST_SEMCOM_STMT =>                                  -- statement descriptor
        SEMCOM_next   :    SAWDUST_SEMCOM_STMT,         --    next statement
        SEMCOM_last   :    SAWDUST_SEMCOM_STMT,         --    previous statement
        SEMCOM_type   :    Short,                       --    statement type
        SEMCOM_index  :    Short,                       --    index for args
        SEMCOM_node   :    ASPEN_NODE,                  --    tree node for args
        SEMCOM_value  :    Univ_Ptr;                    --    value

End

#ifndef SAWDUST_MAIN
#undef PLUM_INCLUDE_ONLY
#endif


/*************************************************************************/
/*                                                                       */
/*      Constants                                                        */
/*                                                                       */
/*************************************************************************/


#define SAWDUST_FONT            WILLOWfontname("PALM_FONT")
```

```
/*****************************************************************************/
/*                                                                           */
/*      Tracing definitions                                                  */
/*                                                                           */
/*****************************************************************************/


extern  Integer SAWDUST__tracelvl;

#define TRACE   if (SAWDUST__tracelvl & SAWDUST_TRACE_ON) SAWDUST_trace
#define ITRACE  if (SAWDUST__tracelvl & SAWDUST_TRACE_INT) SAWDUST_trace
#define DTRACE  if (SAWDUST__tracelvl & SAWDUST_TRACE_DEBUG) SAWDUST_trace



/*****************************************************************************/
/*                                                                           */
/*      Miscellaneous definitions                                            */
/*                                                                           */
/*****************************************************************************/

#define SDE_LOCATE(node,syname,syid) PLUMevent_by_id(SDE__event_current,\
                                                     node,syname,syid)



/*****************************************************************************/
/*                                                                           */
/*      Variable definitions                                                 */
/*                                                                           */
/*****************************************************************************/

extern  ASPEN_NODE      SAWDUST__current_node;
extern  ASPEN_NODE      SAWDUST__parent_node;

extern  Universal       SDE__event_current;



/*****************************************************************************/
/*                                                                           */
/*      Routine definitions                                                  */
/*                                                                           */
/*****************************************************************************/


                                                /* sawdustmain.c        */
extern                  SAWDUST_display_node();
extern                  SAWDUST_scroll();
extern                  SAWDUST_trace();


                                                /* sawdustbutton.c      */
extern                  SAWDUST_button_top();
extern                  SAWDUST_button_bottom();
extern                  SAWDUST_button_in();
extern                  SAWDUST_button_out();
extern                  SAWDUST_button_next();
extern                  SAWDUST_button_back();
extern                  SAWDUST_button_scroll_up();
extern                  SAWDUST_button_scroll_down();
extern                  SAWDUST_button_up();
extern                  SAWDUST_button_down();
extern                  SAWDUST_button_scroll();


/* end of sawdust_local.h */
```

A.5. Program Listing: *sawdustmain.c*

```
/*****************************************************************************/
/*                                                                           */
/*              sawdustmain.c                                                */
/*                                                                           */
/*      Main routines for the Semantic Actions Window                        */
/*                                                                           */
/*****************************************************************************/


#define SAWDUST_MAIN

#include "sawdust_local.h"
#include <sem_reader.h>


/*****************************************************************************/
/*                                                                           */
/*      Local storage definitions                                            */
/*                                                                           */
/*****************************************************************************/

        ASPEN_NODE      SAWDUST__current_node = NULL;
        ASPEN_NODE      SAWDUST__parent_node = NULL;

        Universal       SDE__event_current;

        Integer         SAWDUST__tracelvl = 0;

static  ASH_WINDOW      SAWDUST__window = NULL;
static  Integer         SAWDUST__vtid = -1;

static  Boolean         eolfg = TRUE;
static  Integer         sawdust_font = 0;

static  Integer         num_lines = 0;
static  Integer         num_cols = 0;


/*****************************************************************************/
/*                                                                           */
/*      Forward Definitions                                                  */
/*                                                                           */
/*****************************************************************************/

static                  sawdust_record();
static                  new_sawdust_window();
static                  sawdust_control();
static                  sawdust_sde_current();
static                  setup_sawdust_window();
static                  remove_sawdust_window();
static                  set_window_name();
static                  sawdust_define_scroll();
static                  sawdust_dump();
```

```
/*************************************************************************/
/*                                                                       */
/*      Window Definitions                                               */
/*                                                                       */
/*************************************************************************/


static WILLOW_DEFN sawdust_window = {
   WILLOW_CLASS_USER,
      { "SAWDUST","pecan.icons",'C' },
      { 300,100, 800,1024, 400,300, 1, 1 },
      { ASH_WINDOW_HIT_PARENT,
        WILLOW_TITLE_TAB_SENSE,
        WILLOW_INSTANCE_SAVED_1 },
      new_sawdust_window,
      NULL,
      { { { "IN" },
          WILLOW_LOCATION_TAIL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_in) },
        { { "OUT" },
          WILLOW_LOCATION_TAIL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_out) },
        { { "NEXT" },
          WILLOW_LOCATION_TAIL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_next) },
        { { "BACK" },
          WILLOW_LOCATION_TAIL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_back) },
        { { "TOP" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_top) },
        { { "BOTTOM" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_bottom) },
        { { "SCROLL UP" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_scroll_up) },
        { { "SCROLL DOWN" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_scroll_down) },
        { { "UP" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_up) },
        { { "DOWN" },
          WILLOW_LOCATION_BOTTOM,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_USER(SAWDUST_button_down) },
        { { "SCROLL" },
          WILLOW_LOCATION_R,
          WILLOW_BUTTON_REGION,
          WILLOW_ACTION_USER(SAWDUST_button_scroll) },
        { { "Move", "pecan.icons", '1', 0, 0, 1 },
          WILLOW_LOCATION_UL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_MOVE(DEFAULT) },
        { { "Size", "pecan.icons", '0', 0, 0, 1 },
          WILLOW_LOCATION_UR,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_TYPE_AUX9, NULL, WILLOW_ACTION_DEFAULT },
```

```
        { { "Remove" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_ICON(DEFAULT) },
        { { "Push" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_PUSH(DEFAULT) },
        { { "Pop" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_POP(DEFAULT) }
      }
};


/*****************************************************************************/
/*                                                                           */
/*      SAWDUSTinit -- initialize sawdust display                            */
/*                                                                           */
/*****************************************************************************/


SAWDUSTinit()
{
   TRACE("SAWDUSTinit");

   SAWDUST__window = NULL;
   SAWDUST__vtid = NULL;
   SAWDUST__current_node = NULL;
   eolfg = TRUE;
   num_lines = 0;
   num_cols = 0;

   WILLOWdefine_window(&sawdust_window);

   SDE__event_current = PLUMinq_event_id("SDE_$CURRENT");

   PLUMaccept_event(sawdust_sde_current,"SDE_$CURRENT");
};


/*****************************************************************************/
/*                                                                           */
/*      SAWDUSTtrace -- set trace flag                                       */
/*                                                                           */
/*****************************************************************************/


SAWDUSTtrace(lvl)
   Integer lvl;
{
   TRACE("SAWDUSTtrace");

   SAWDUST__tracelvl = lvl;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_display_node -- display current node                          */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_display_node()
{
   register String header;

   ITRACE("SAWDUST_display_node");

   VT$PUSH(SAWDUST__vtid);
   VT$MOVE(0,0);
   VT$ERASE_SCREEN;
   VT$POP;

   SAWDUST_scroll(0,0,0);

   if (SAWDUST__current_node != NULL) {
      if ((SAWDUST__parent_node = ASPENinq_parent(SAWDUST__current_node))
          == NULL) {
         header = "'%s' - PARENT DOES NOT EXIST - children: %d";
         sawdust_record(header,ASPENrule_name
                                (ASPENinq_rule(SAWDUST__current_node)),
                               ASPENinq_arity(SAWDUST__current_node));
       }
      else {
         header = "'%s' - Child number: %d of %d - children: %d";
         sawdust_record(header,
                        ASPENrule_name
                         (ASPENinq_rule(SAWDUST__current_node)),
                        ASPENinq_son_number(SAWDUST__current_node) + 1,
                        ASPENinq_arity(SAWDUST__parent_node),
                        ASPENinq_arity(SAWDUST__current_node));
       };
    };

   sawdust_record
          ("-------------------------------------------------------------------");

   if (SAWDUST__current_node == NULL) {
      sawdust_record("NULL");
    }
   else {
      sawdust_dump();
    };

   sawdust_record
          ("-------------------------------------------------------------------");

   sawdust_define_scroll();
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_scroll -- scroll VT window                                   */
/*                                                                           */
/*****************************************************************************/


SAWDUST_scroll(dl,dc,abs)
   Integer dl,dc;
   Integer abs;
{
   Integer rl,rc;
   Integer cl,cc;
   register Integer b;

   ITRACE("SAWDUST_scroll %d %d %d",dl,dc,abs);

   VT$PUSH(SAWDUST__vtid);
   VT$NO_SCROLL;
   VT$INQ_REGION(&rl,&rc);

   if (dl != 0) {
      rl += dl*(num_lines/4);
    }
   else if (dc != 0) {
      rc += dc*(num_cols/4);
    }
   else {
      VT$INQ_CURRENT(&cl,&cc);
      ITRACE("\tscroll absolute %d %d %d %d %d",rl,rc,cl,cc,abs);
      b = MAX(cl,rl+num_lines);
      b = abs*b/100-num_lines/2;
      if (b < 0) b = 0;
      else if (b > cl) b = cl;
      rl = b;
      rc = 0;
    };

   ITRACE("\tscroll to %d %d",rl,rc);

   VT$REGION(rl,rc);
   VT$POP;

   sawdust_define_scroll();
};


/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_trace -- output trace information                            */
/*                                                                           */
/*****************************************************************************/


SAWDUST_trace(msg,a1,a2,a3,a4,a5,a6,a7,a8,a9)
   String msg;
   Integer a1,a2,a3,a4,a5,a6,a7,a8,a9;
{
   Character mbf[1024];

   sprintf(mbf,msg,a1,a2,a3,a4,a5,a6,a7,a8,a9);

   printf("SAWDUST: %s\n",mbf);
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      sawdust_record -- put message in transcript for Semantic             */
/*                        Actions Window                                     */
/*                                                                           */
/*****************************************************************************/


static
sawdust_record(msg,a1,a2,a3,a4)
   String msg;
   Universal a1,a2,a3,a4;
{
   Character buf[256],buf1[256];

   DTRACE("sawdust_record %s",msg);

   VT$PUSH(SAWDUST__vtid);
   VT$NO_SCROLL;
   VT$FONT(sawdust_font);

   if (!eolfg) VT$OUT("\n");

   sprintf(buf,msg,a1,a2,a3,a4);
   sprintf(buf1,"%s\n",buf);
   VT$OUT(buf1);

#ifdef VAX
   printf("%s",buf1);
#endif

   VT$POP;

   eolfg = TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      new_sawdust_window -- set up sawdust display window                  */
/*                                                                           */
/*****************************************************************************/


static
new_sawdust_window()
{
   register ASH_WINDOW w;

   DTRACE("new_sawdust_window");

   w = ASHinq_window();

   SAWDUST__window = w;

   ASHset_control(sawdust_control);

   setup_sawdust_window();
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      sawdust_control — control message interpreter                        */
/*                                                                           */
/*****************************************************************************/


static
sawdust_control(msg,w)
   String msg;
   ASH_WINDOW w;
{
   DTRACE("sawdust_control %s 0x%x",msg,w);

   if (STREQL(msg,"PDS$NEXT")) return ASH_CONTROL_OK;

   if (STREQL(msg,"ASH$RESIZE")) {
      setup_sawdust_window();
    }
   else if (STREQL(msg,"ASH$INQ_RESIZE")) {
      remove_sawdust_window();
    }
   else if (STREQL(msg,"ASH$REMOVE")) {
      remove_sawdust_window();
      SAWDUST__window = NULL;
      SAWDUST__current_node = NULL;
    };

   return ASH_CONTROL_REJECT;
};



/*****************************************************************************/
/*                                                                           */
/*      sawdust_sde_current — handle sde_locate event                        */
/*                                                                           */
/*****************************************************************************/


static
sawdust_sde_current(evt,act,node,name,id)
   String evt;
   PLUM_EVENT_ACTION act;
   ASPEN_NODE node;
   String name;
   Integer id;
{
   DTRACE("sawdust_sde_locate %d 0x%x %s",act,node,name);

   if (act != PLUM_EVENT_DO) return;

   if ((SAWDUST__window != NULL) && (name != "SAWDUST") &&
       (SAWDUST__current_node != node)) {

      SAWDUST__current_node = node;

      set_window_name();

      SAWDUST_display_node();
    }
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      setup_sawdust_window -- set up sawdust display window                */
/*                                                                           */
/*****************************************************************************/


static
setup_sawdust_window()
{
   DTRACE("setup_sawdust_window");

   ASHpush_window();

   ASHselect(SAWDUST__window);

   SAWDUST__vtid = VTopen();
   VT$PUSH(SAWDUST__vtid);
   VT$SCROLL;

   sawdust_font = VT$LOADFONT(SAWDUST_FONT);
   VT$FONT(sawdust_font);
   VT$INQ_SIZE(&num_lines,&num_cols);

   VT$POP;

   ASHpop_window();

   set_window_name();

   SAWDUST_display_node();
};



/*****************************************************************************/
/*                                                                           */
/*      remove_sawdust_window -- remove sawdust display window               */
/*                                                                           */
/*****************************************************************************/


static
remove_sawdust_window()
{
   DTRACE("remove_sawdust_window");

   VTclose(SAWDUST__vtid);
};
```

```
/************************************************************************/
/*                                                                      */
/*      set_window_name -- put proper name in title for Semantic        */
/*                         Actions Window                               */
/*                                                                      */
/************************************************************************/


static
set_window_name()
{
   Character buf[256];

   DTRACE("set_window_name");

   if (SAWDUST__window != NULL) {
      ASHpush_window();
      ASHselect(SAWDUST__window);
      if (SAWDUST__current_node == NULL) {
         strcpy(buf,"Semantic actions");
       }
      else {
         sprintf(buf,"Semantic actions for %s",
                 ASPENinq_name(SAWDUST__current_node));
       };
      ASHset_window_name(buf);
      ASHpop_window();
    };
};


/************************************************************************/
/*                                                                      */
/*      sawdust_define_scroll -- define scroll region                   */
/*                                                                      */
/************************************************************************/


static
sawdust_define_scroll()
{
   Integer rl,rc;
   Integer cl,cc;
   register Integer b;

   DTRACE("sawdust_define_scroll");

   VT$PUSH(SAWDUST__vtid);
   VT$INQ_CURRENT(&cl,&cc);
   VT$INQ_REGION(&rl,&rc);
   VT$POP;

   b = MAX(cl,rl+num_lines);
   WILLOWbutton_feedback(SAWDUST__window,"SCROLL",TRUE,
                         WILLOW_SCROLL_REGION(rl*100/b,
                                              (rl+num_lines)*100/b));
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      sawdust_dump -- dump semantic statements between head and tail       */
/*                      of parent node                                       */
/*                                                                           */
/*****************************************************************************/


static
sawdust_dump()
{
   SAWDUST_SEMCOM_STMT current_head,current_tail,parent_head,parent_tail;
   register SAWDUST_SEMCOM_STMT s,ls;
   String indent;

   DTRACE("sawdust_dump");

   ASPENinq_semantics(SAWDUST__current_node,
                      &current_head, &current_tail);

   if (SAWDUST__parent_node != NULL) {
      ASPENinq_semantics(SAWDUST__parent_node,
                         &parent_head, &parent_tail);
    }
   else
    {
      parent_head = current_head;
      parent_tail = current_tail;
    }

   indent = "";

   if (parent_head != NULL) ls = parent_head -> SEMCOM_last;

   for (s = parent_head; s != NULL;
        s = (s == parent_tail ? NULL : s -> SEMCOM_next)) {

      if (s == current_head) {
         sawdust_record
          ("-----------------------------------------------------------------");
         indent = "-> ";
       }

      sawdust_record("%s(0x%x) :\t%s %d\t[0x%x]\t@0x%x",
                     indent,s,SEMDATATABLE[s->SEMCOM_type].SEM_stmt_name,
                     s->SEMCOM_index,s->SEMCOM_value,s->SEMCOM_node);

      if (s->SEMCOM_last != ls) sawdust_record("\t***BAD LAST 0x%x",
                                               s->SEMCOM_last);
      ls = s;

      if (s == current_tail) {
         sawdust_record
          ("-----------------------------------------------------------------");
         indent = "";
       }
    };
};


/* end of sawdustmain.c */
```

A.6. Program Listing: *sawdustbutton.c*

```
/*****************************************************************************/
/*                                                                           */
/*              sawdustbutton.c                                              */
/*                                                                           */
/*      Button handling routines for the Semantic Actions Window             */
/*                                                                           */
/*****************************************************************************/

#include "sawdust_local.h"

/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_top -- handle TOP button                              */
/*                                                                           */
/*****************************************************************************/

int
SAWDUST_button_top(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_top %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(0,0,0);

   return TRUE;
};


/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_bottom -- handle BOTTOM button                        */
/*                                                                           */
/*****************************************************************************/

int
SAWDUST_button_bottom(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_bottom %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(0,0,100);

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_in -- handle IN button                                */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_in(dir)
   WILLOW_ACTION_MODE dir;
{
   register ASPEN_NODE s;

   ITRACE("SAWDUST_button_in %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   if ((SAWDUST__current_node != NULL) &&
       ((s = ASPENinq_son(SAWDUST__current_node,0)) != NULL)) {

      SAWDUST__current_node = s;

      SDE_LOCATE(SAWDUST__current_node,"SAWDUST",0);

      SAWDUST_display_node();
    };

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_out -- handle OUT button                              */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_out(dir)
   WILLOW_ACTION_MODE dir;
{
   register ASPEN_NODE p;

   ITRACE("SAWDUST_button_out %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   if ((SAWDUST__current_node != NULL) &&
       ((p = ASPENinq_parent(SAWDUST__current_node)) != NULL)) {

      SAWDUST__current_node = p;

      SDE_LOCATE(SAWDUST__current_node,"SAWDUST",0);

      SAWDUST_display_node();
    };

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_next -- handle NEXT button                            */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_next(dir)
   WILLOW_ACTION_MODE dir;
{
   register Integer s;
   register ASPEN_NODE p;

   ITRACE("SAWDUST_button_next %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   if ((SAWDUST__current_node != NULL) &&
       ((p = ASPENinq_parent(SAWDUST__current_node)) != NULL) &&
       ((s = ASPENinq_son_number(SAWDUST__current_node)) <
        ASPENinq_arity(p)-1)) {

      SAWDUST__current_node = ASPENinq_son(p,s+1);

      SDE_LOCATE(SAWDUST__current_node,"SAWDUST",0);

      SAWDUST_display_node();
    };

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_back -- handle BACK button                            */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_back(dir)
   WILLOW_ACTION_MODE dir;
{
   register ASPEN_NODE p;
   register Integer s;

   ITRACE("SAWDUST_button_back %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   if ((SAWDUST__current_node != NULL) &&
       ((s = ASPENinq_son_number(SAWDUST__current_node)) > 0) &&
       ((p = ASPENinq_parent(SAWDUST__current_node)) != NULL)) {

      SAWDUST__current_node = ASPENinq_son(p,s-1);

      SDE_LOCATE(SAWDUST__current_node,"SAWDUST",0);

      SAWDUST_display_node();
    };

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_scroll_up -- handle SCROLL UP button                  */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_scroll_up(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_scroll_up %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(-4,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_scroll_down -- handle SCROLL DOWN button              */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_scroll_down(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_scroll_down %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(4,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_up -- handle UP button                                */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_up(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_up %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(-1,0,0);

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_down -- handle DOWN button                            */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_down(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_down %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(1,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SAWDUST_button_scroll -- handle scroll bar                           */
/*                                                                           */
/*****************************************************************************/


int
SAWDUST_button_scroll(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SAWDUST_button_scroll %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SAWDUST_scroll(0,0,WILLOWinq_scroll());

   return TRUE;
};

/* end of sawdustbutton.c */
```

Appendix B
The SEMCOM Module

As explained in Chapter 5, the SEMCOM module handles incremental compilation in PECAN. This appendix contains a description of, and selected program listings from, the files that make up that module as modified in the manner described in Chapter 6.

B.1. The Compilation Monitor

The SEMCOM module has been modified so as to provide a new view; a window which displays compilation information. Buttons are provided which allow the window to be scrolled so that all of the information may be examined. Other buttons allow the programmer to choose the method of compilation to be employed when a modification is made to the AST.

An example PECAN screen, showing the compilation monitor, is reproduced in Figure B-1. The SDE and the flow view are on the left side of the screen, the compilation monitor is on the right. The scroll buttons are identical to those provided by the semantic actions view, and explained in §A.1. In addition, the CLEAR button clears the screen, erasing any information which may have been displayed on it.

The INCREMENTAL, PROCEDURE and COMPLETE buttons choose the compilation method that will be next used. Each choice is echoed on the screen when made. The COMPILE button forces SEMCOM to compile immediately (unless the compilation method is incremental). The AUTO button toggles automatic recompilation. When automatic recompilation is set, compilation is triggered by every change made to the AST. When automatic recompilation is not set,[1] compilation is not performed unless the COMPILE button is used or (if procedure compilation is selected) a change is made to the AST outside the procedure within which the last change was made.

---

[1]The AUTO button becomes the MANUAL button when automatic recompilation is not set in order to display the state of automatic recompilation.

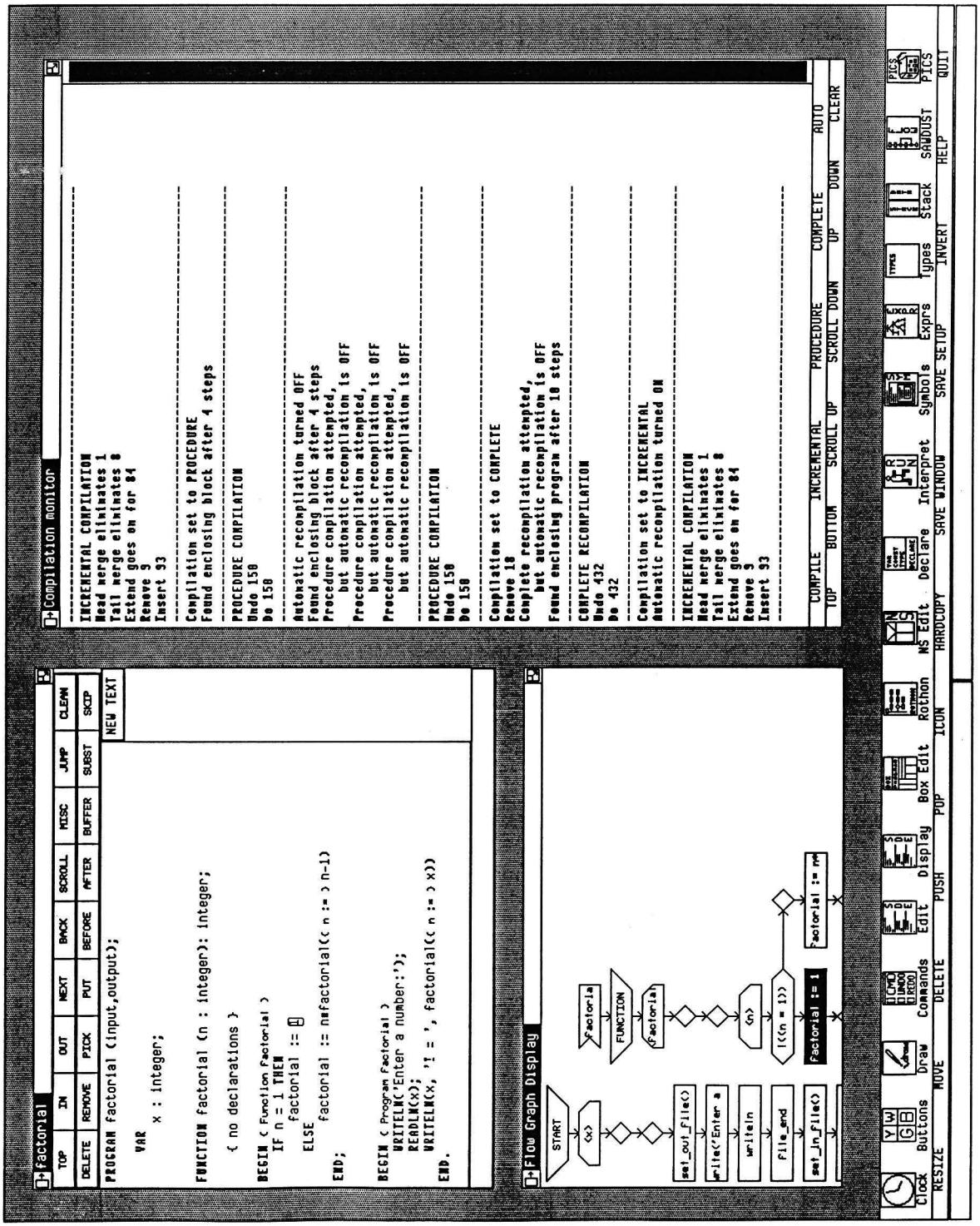

Figure B-1: The Compilation Monitor

B.2. Implementation Details

The modified SEMCOM module consists of eight files:

semcom.h (§B.3)
The external header file.
Lists the externally accessible SEMCOM functions and gives details of the module's trace facilities.

semcom_local.hi (§B.4)
The local header file.
Includes the definition of the SEMCOM_STMT type.

semcommain.c (Not listed - modifications to SEMCOM did not significantly affect this file.)
Includes the initialization and trace routines, and *sem_event_node* which is invoked by PLUM when an ASPEN_$NODE_CHANGE event is broadcast.

semcomstmt.c (§B.5)
Maintains lists of SEMCOM_STMTs. Includes the *_SEMCOM_replace_list* and *_SEMCOM_remove_list* functions (modified to handle procedure compilation and complete compilation) and the new functions *SEMCOM_force_compilation* (which implements the COMPILE button), *copy_list* (which makes a copy of an existing list of SEMCOM_STMTs), and *enclosing_block* and *enclosing_program* (which find the enclosing node of the appropriate type in the AST).

semcomeval.c (§B.6)
Contains the *head_merge*, *tail_merge*, *extend* and *insert* functions. The *remove* function is renamed to *SEMCOM_remove* (because the modifications required that it be visible to other files in the SEMCOM module). These low-level functions are called by the new functions *SEMCOM_change_incremental*, *SEMCOM_change_procedure* and *SEMCOM_change_complete* which replace the function *_SEMCOM_change*.

semcomexec.c (Not listed - modifications to SEMCOM did not affect this file.)
Handles the execution and unexecution of SEMCOM_STMTs. *_SEMCOM_execute* and *_SEMCOM_unexecute* both use a large *switch* statement with a case for each type of SEMCOM_STMT. Maintains and modifies the values of the current items (using *_SEMCOM_set_current*, *_SEMCOM_get_currents*, *etc.*).

semcomwindow.c (§B.7 - New file)
Defines the SEMCOM compilation monitor (using the WILLOW module) and controls the display of information on that screen.

semcombutton.c (§B.8 - New file)
Button handling routines.

Only those functions that were added or altered when the SEMCOM module was modified have been included in the program listings that follow.

B.3. Program Listing: *semcom.h*

```
/************************************************************************/
/*                                                                      */
/*              semcom.h                                                */
/*                                                                      */
/*      External definitions for using incremental symbol compiler      */
/*                                                                      */
/************************************************************************/
/*      Copyright 1984 Brown University -- Steven P. Reiss              */
/*      Modified  James Popple  September/October 1987                  */



/************************************************************************/
/*                                                                      */
/*      Tracing definitions -- use "-Dsn" switch, where n gives         */
/*                             the type(s) of trace.                    */
/*                                                                      */
/************************************************************************/

#define SEMCOM_TRACE_OFF        0
#define SEMCOM_TRACE_ON         1
#define SEMCOM_TRACE_INT        2
#define SEMCOM_TRACE_DEBUG      4
#define SEMCOM_TRACE_DUMP       8
#define SEMCOM_TRACE_COMPILE    16



/************************************************************************/
/*                                                                      */
/*      General routines                                                */
/*                                                                      */
/************************************************************************/

extern                  SEMCOMinit();
extern                  SEMCOMtrace();
extern                  SEMCOMupdate();
extern  SYM_REF         SEMCOMinq_ref();
extern  EXPR_NODE       SEMCOMinq_expr();
extern  TYPE_DEF        SEMCOMinq_type();
extern  FLOW_NODE       SEMCOMinq_flow();

extern  Integer         SEMCOMsuggest_text();
extern  Boolean         SEMCOMtest_begin();



/* end of semcom.h */
```

B.4. Program Listing: *semcom_local.hi*

```
/*************************************************************************/
/*                                                                       */
/*              semcom_local.h  (derived from semcom_local.hi)           */
/*                                                                       */
/*      Local definitions for incremental symbol compiler                */
/*                                                                       */
/*************************************************************************/


#include <plum.h>
#include <aspen.h>
#include <symbols.h>
#include <vt.h>
#include <willow.h>
#include <type.h>
#include <expr.h>
#include <flow.h>
#include "semcom.h"
#include <ash.h>



/*************************************************************************/
/*                                                                       */
/*      Data structures                                                  */
/*                                                                       */
/*************************************************************************/


#ifndef SEMCOM_MAIN
#define PLUM_INCLUDE_ONLY
#endif


Mode SemcomDefs Is

Type ASPEN_NODE From Mode Aspen External;
Type SYM_SCOPE From Mode Symbols External;
Type SYM_NAME From Mode Symbols External;
Type SYM_OBJECT From Mode Symbols External;
Type SYM_OBJECTSET From Mode Symbols External;
Type SYM_REF From Mode Symbols External;
Type FLOW_NODE From Mode Flows External;
Type TYPE_DEF From Mode Types External;
Type EXPR_NODE From Mode Exprs External;


Type SEMCOM_COMPILATION_TYPE Is Enum
   SEMCOM_COMP_INCREMENTAL,
   SEMCOM_COMP_PROCEDURE,
   SEMCOM_COMP_COMPLETE;


SEMCOM_NAME_INFO =>                                 — info on name basis
        SEMCOM_name     :       String,             —    the name itself
        SEMCOM_first    :       SEMCOM_STMT,        —    first symbol stmt
        SEMCOM_active   :       Boolean;            —    name is active


SEMCOM_STMT =>                                      — statement descriptor
        SEMCOM_next     :       SEMCOM_STMT,        —    next statement
        SEMCOM_last     :       SEMCOM_STMT,        —    previous statement
        SEMCOM_type     :       Short,              —    statement type
        SEMCOM_index    :       Short,              —    index for args
        SEMCOM_node     :       ASPEN_NODE,         —    tree node for args
        SEMCOM_value    :       Univ_Ptr;           —    value
```

```
SEMCOM_CUR =>                                          -- current values
        SEMCOM_cur_scope   :     SYM_SCOPE,
        SEMCOM_cur_ref     :     SYM_REF,
        SEMCOM_cur_flow    :     FLOW_NODE,
        SEMCOM_cur_type    :     TYPE_DEF,
        SEMCOM_cur_expr    :     EXPR_NODE,
        SEMCOM_cur_use_scope :   SYM_SCOPE,
        SEMCOM_cur_build_type :  TYPE_DEF,
        SEMCOM_cur_mode    :     Integer;

End


#ifdef PLUM_INCLUDE_ONLY
#undef PLUM_INCLUDE_ONLY
#endif



/*****************************************************************************/
/*                                                                           */
/*      Constants                                                            */
/*                                                                           */
/*****************************************************************************/

#define SEMCOM_FONT     WILLOWfontname("PALM_FONT")



/*****************************************************************************/
/*                                                                           */
/*      Tracing definitions                                                  */
/*                                                                           */
/*****************************************************************************/

extern  Integer         _SEMCOM__trace_level;
extern  Boolean         _SEMCOM__initfg;

#define TRACE           if (_SEMCOM__trace_level & SEMCOM_TRACE_ON)\
                            _SEMCOM_trace
#define ITRACE          if (_SEMCOM__trace_level & SEMCOM_TRACE_INT)\
                            _SEMCOM_trace
#define DTRACE          if (_SEMCOM__trace_level & SEMCOM_TRACE_DEBUG)\
                            _SEMCOM_trace
#define CTRACE          if (_SEMCOM__trace_level & SEMCOM_TRACE_COMPILE)\
                            _SEMCOM_trace

#define ERROR(msg)      _SEMCOM_trace("Error: msg")
#define ABORT(msg)      (_SEMCOM_trace("ABORT: msg"), abort())

#define CHECKINIT       if (!_SEMCOM__initfg) SEMCOMinit()
#define ENTER           CHECKINIT; TRACE
```

```
/*****************************************************************************/
/*                                                                           */
/*      Miscellaneous definitions                                            */
/*                                                                           */
/*****************************************************************************/

#define CUR_NAME        (_SEMCOM__cur->SEMCOM_name)
#define FIRSTSTMT       (_SEMCOM__cur->SEMCOM_first)
#define ACTIVE          (_SEMCOM__cur->SEMCOM_active)



/*****************************************************************************/
/*                                                                           */
/*      Variable definitions                                                 */
/*                                                                           */
/*****************************************************************************/

extern  ASH_WINDOW              SEMCOM__window;
extern  Boolean                 SEMCOM__auto_recomp;
extern  SEMCOM_COMPILATION_TYPE SEMCOM__comp_type;



/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcommain.c                                  */
/*                                                                           */
/*****************************************************************************/

extern  SEMCOM_NAME_INFO _SEMCOM__cur;

extern                  _SEMCOM_trace();
extern                  _SEMCOM_dump();
extern                  _SEMCOM_set_current_name();
extern                  _SEMCOM_reset_current_name();
extern                  _SEMCOM_set_current_node();



/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcomstmt.c                                  */
/*                                                                           */
/*****************************************************************************/

extern                  _SEMCOM_stmt_init();
extern  Boolean         SEMCOM_test_for();
extern  Boolean         SEMCOM_test_del_ok();
extern                  _SEMCOM_replace_list();
extern                  _SEMCOM_remove_list();
extern                  SEMCOM_force_compilation();
extern                  _SEMCOM_stmt_free();
extern  SEMCOM_STMT     _SEMCOM_findprevious();
```

```
/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcomeval.c                                  */
/*                                                                           */
/*****************************************************************************/

extern                  _SEMCOM_eval_init();
extern                  SEMCOM_change_incremental();
extern                  SEMCOM_change_procedure();
extern                  SEMCOM_change_complete();
extern                  SEMCOM_remove();


/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcomexec.c                                  */
/*                                                                           */
/*****************************************************************************/

extern                  _SEMCOM_exec_init();
extern                  _SEMCOM_set_current();
extern                  _SEMCOM_unexecute();
extern                  _SEMCOM_execute();
extern                  _SEMCOM_get_currents();
extern                  SEMCOM_free_value();


/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcomwindow.c                                */
/*                                                                           */
/*****************************************************************************/

extern                  SEMCOM_window_init();
extern                  SEMCOM_scroll();
extern                  SEMCOM_clear_screen();
extern                  SEMCOM_record();


/*****************************************************************************/
/*                                                                           */
/*      Local definitions from semcombutton.c                                */
/*                                                                           */
/*****************************************************************************/

extern                  SEMCOM_button_compile();
extern                  SEMCOM_button_incremental();
extern                  SEMCOM_button_procedure();
extern                  SEMCOM_button_complete();
extern                  SEMCOM_button_auto();
extern                  SEMCOM_button_top();
extern                  SEMCOM_button_bottom();
extern                  SEMCOM_button_scroll_up();
extern                  SEMCOM_button_scroll_down();
extern                  SEMCOM_button_up();
extern                  SEMCOM_button_down();
extern                  SEMCOM_button_clear();
extern                  SEMCOM_button_scroll();


/* end of semcom_local.h */
```

B.5. Abridged Program Listing: *semcomstmt.c*

```
/*****************************************************************************/
/*                                                                           */
/*              semcomstmt.c                                                 */
/*                                                                           */
/*      Routines for statement list maintenance in symbol compiler           */
/*                                                                           */
/*****************************************************************************/


#include "semcom_local.h"
#include <sem_reader.h>


/*****************************************************************************/
/*                                                                           */
/*      Local storage                                                        */
/*                                                                           */
/*****************************************************************************/


        SEMCOM_NAME_INFO _SEMCOM__cur;

static  SEMCOM_STMT     freelist = NULL;
static  SEMCOM_STMT     firststmt = NULL;

static  SEMCOM_STMT     proc_old_hd = NULL;
static  SEMCOM_STMT     proc_old_tl = NULL;
static  SEMCOM_STMT     proc_new_hd = NULL;
static  SEMCOM_STMT     proc_new_tl = NULL;

static  SEMCOM_STMT     proc_after = NULL;

static  ASPEN_NODE      current_node = NULL;
static  ASPEN_NODE      previous_node = NULL;
static  ASPEN_NODE      current_procedure = NULL;


/*****************************************************************************/
/*                                                                           */
/*      Miscellaneous definitions                                            */
/*                                                                           */
/*****************************************************************************/


#define CUR_MAX_STMT    2
#define BLOCK           19


/*****************************************************************************/
/*                                                                           */
/*      Forward Definitions                                                  */
/*                                                                           */
/*****************************************************************************/


static                  semcom_new_node();
static  SEMCOM_STMT     stmt_list();
static  SEMCOM_STMT     eval_do_stmt();
static  SEMCOM_STMT     new_stmt();
static                  copy_list();
static  ASPEN_NODE      enclosing_block();
static  ASPEN_NODE      enclosing_program();
static  int             varcmp();
```

```
/*************************************************************************/
/*                                                                       */
/*      _SEMCOM_stmt_init -- initialize module                           */
/*                                                                       */
/*************************************************************************/

_SEMCOM_stmt_init()
{
   ITRACE("_SEMCOM_stmt_init");

   freelist = NULL;
   firststmt = NULL;

   PLUMaccept_event(semcom_new_node,"SDE_$CURRENT");
};
```

Not listed - *SEMCOMsuggest_text*

Not listed - *SEMCOMtest_begin*

Not listed - *SEMCOM_test_for*

Not listed - *SEMCOM_test_del_ok*

```
/*************************************************************************/
/*                                                                       */
/*      _SEMCOM_replace_list -- replace the statement list for a node    */
/*                                                                       */
/*************************************************************************/

_SEMCOM_replace_list(n)
   ASPEN_NODE n;
{
   register SEMCOM_STMT s;
   SEMCOM_STMT b,olds,oldtl,hd,tl;

   ITRACE("_SEMCOM_replace_list 0x%x",n);

   switch (SEMCOM__comp_type) {

      case SEMCOM_COMP_INCREMENTAL:

         b = _SEMCOM_findprevious(n);

         ASPENinq_semantics(n,&olds,&oldtl);

         ASPENset_semantics(n,NULL,NULL);

         if (ACTIVE) stmt_list(n,NULL);

         ASPENinq_semantics(n,&hd,&tl);

         SEMCOM_change_incremental(b,olds,oldtl,hd,tl);

         break;
```

```
case SEMCOM_COMP_PROCEDURE:

   if (enclosing_block(n,FALSE) == current_procedure) {

      ASPENinq_semantics(n,&olds,&oldtl);

      ASPENset_semantics(n,NULL,NULL);

      if (ACTIVE) stmt_list(n,NULL);

      ASPENinq_semantics(n,&hd,&tl);

      if ((hd != NULL) && (olds != NULL)) {
         hd->SEMCOM_last = olds->SEMCOM_last;
         if (olds->SEMCOM_last != NULL)
            olds->SEMCOM_last->SEMCOM_next = hd;
       };

      if ((tl != NULL) && (oldtl != NULL)) {
         tl->SEMCOM_next = oldtl->SEMCOM_next;
         if (oldtl->SEMCOM_next != NULL)
            oldtl->SEMCOM_next->SEMCOM_last = tl;
       }
    }
   else {
      if ((current_procedure != NULL) &&
          (proc_new_hd != NULL) &&
          (proc_new_tl != NULL) &&
          (proc_old_hd != NULL) &&
          (proc_old_tl != NULL)) {
         SEMCOM_record("Moved out of previous procedure,");
         SEMCOM_record
                   ("   forcing compilation of previous procedure");
         SEMCOM_force_compilation();
       }

      current_procedure = enclosing_block(n,TRUE);

      proc_after = _SEMCOM_findprevious(current_procedure);

      ASPENinq_semantics(current_procedure,&proc_new_hd,
                         &proc_new_tl);

      copy_list(proc_new_hd,proc_new_tl,&proc_old_hd,&proc_old_tl);

      ASPENinq_semantics(n,&olds,&oldtl);

      ASPENset_semantics(n,NULL,NULL);

      if (ACTIVE) stmt_list(n,NULL);

      ASPENinq_semantics(n,&hd,&tl);

      if ((hd != NULL) && (olds != NULL)) {
         hd->SEMCOM_last = olds->SEMCOM_last;
         if (olds->SEMCOM_last != NULL)
            olds->SEMCOM_last->SEMCOM_next = hd;
       };

      if ((tl != NULL) && (oldtl != NULL)) {
         tl->SEMCOM_next = oldtl->SEMCOM_next;
         if (oldtl->SEMCOM_next != NULL)
            oldtl->SEMCOM_next->SEMCOM_last = tl;
       }
   }
```

```
            if (SEMCOM__auto_recomp) {

               SEMCOM_change_procedure(proc_after,proc_old_hd,proc_old_tl,
                                       proc_new_hd,proc_new_tl);
               proc_old_hd = NULL;
               proc_old_tl = NULL;
               proc_new_hd = NULL;
               proc_new_hd = NULL;
               current_procedure = NULL;
               proc_after = NULL;
             }

            else

               SEMCOM_change_procedure(NULL,NULL,NULL,NULL,NULL);

            break;

         case SEMCOM_COMP_COMPLETE:

            ASPENinq_semantics(n,&olds,&oldtl);

            ASPENset_semantics(n,NULL,NULL);

            if (ACTIVE) stmt_list(n,NULL);

            ASPENinq_semantics(n,&hd,&tl);

            if ((hd != NULL) && (olds != NULL)) {
               hd->SEMCOM_last = olds->SEMCOM_last;
               if (olds->SEMCOM_last != NULL)
                  olds->SEMCOM_last->SEMCOM_next = hd;
             };

            if ((tl != NULL) && (oldtl != NULL)) {
               tl->SEMCOM_next = oldtl->SEMCOM_next;
               if (oldtl->SEMCOM_next != NULL)
                  oldtl->SEMCOM_next->SEMCOM_last = tl;
             };

            SEMCOM_remove(olds,oldtl);

            if (SEMCOM__auto_recomp) {

               n = enclosing_program(n,TRUE);

               ASPENinq_semantics(n,&hd,&tl);

               SEMCOM_change_complete(hd,tl);
             }

            else

               SEMCOM_change_complete(NULL,NULL);

            break;
       };

      if (_SEMCOM__trace_level & SEMCOM_TRACE_DUMP) _SEMCOM_dump();
   };
```

```
/*************************************************************************/
/*                                                                       */
/*      _SEMCOM_remove_list -- remove the statement list for a node      */
/*                                                                       */
/*************************************************************************/


_SEMCOM_remove_list(n)
   ASPEN_NODE n;
{
   register SEMCOM_STMT s;
   SEMCOM_STMT b,olds,oldtl,hd,tl;

   ITRACE("_SEMCOM_remove_list 0x%x",n);

   switch (SEMCOM__comp_type) {

      case SEMCOM_COMP_INCREMENTAL:

         b = _SEMCOM_findprevious(n);

         ASPENinq_semantics(n,&olds,&oldtl);

         ASPENset_semantics(n,NULL,NULL);

         SEMCOM_change_incremental(b,olds,oldtl,NULL,NULL);

         break;

      case SEMCOM_COMP_PROCEDURE:

         if (enclosing_block(n,FALSE) == current_procedure) {

            ASPENinq_semantics(n,&olds,&oldtl);

            ASPENset_semantics(n,NULL,NULL);

            if ((olds != NULL) && (olds->SEMCOM_last != NULL) &&
                (oldtl != NULL))
               olds->SEMCOM_last->SEMCOM_next = oldtl->SEMCOM_next;

            if ((oldtl != NULL) && (oldtl->SEMCOM_next != NULL) &&
                (olds != NULL))
               oldtl->SEMCOM_next->SEMCOM_last = olds->SEMCOM_last;

            SEMCOM_remove(olds,oldtl);
          }
         else {
            if ((current_procedure != NULL) &&
                (proc_new_hd != NULL) &&
                (proc_new_tl != NULL) &&
                (proc_old_hd != NULL) &&
                (proc_old_tl != NULL)) {
               SEMCOM_record("Moved out of previous procedure,");
               SEMCOM_record
                         ("   forcing compilation of previous procedure");
               SEMCOM_force_compilation();
             }

            current_procedure = enclosing_block(n,TRUE);

            proc_after = _SEMCOM_findprevious(current_procedure);

            ASPENinq_semantics(current_procedure,&proc_new_hd,
                               &proc_new_tl);

            copy_list(proc_new_hd,proc_new_tl,&proc_old_hd,&proc_old_tl);

            ASPENinq_semantics(n,&olds,&oldtl);
```

```
            ASPENset_semantics(n,NULL,NULL);

            if ((olds != NULL) && (olds->SEMCOM_last != NULL) &&
                (oldtl != NULL))
               olds->SEMCOM_last->SEMCOM_next = oldtl->SEMCOM_next;

            if ((oldtl != NULL) && (oldtl->SEMCOM_next != NULL) &&
                (olds != NULL))
               oldtl->SEMCOM_next->SEMCOM_last = olds->SEMCOM_last;

            SEMCOM_remove(olds,oldtl);
          }

          if (SEMCOM__auto_recomp) {

            SEMCOM_change_procedure(proc_after,proc_old_hd,proc_old_tl,
                                    proc_new_hd,proc_new_tl);
            proc_old_hd = NULL;
            proc_old_tl = NULL;
            proc_new_hd = NULL;
            proc_new_hd = NULL;
            current_procedure = NULL;
            proc_after = NULL;
          }

          else

            SEMCOM_change_procedure(NULL,NULL,NULL,NULL,NULL);

          break;

        case SEMCOM_COMP_COMPLETE:

          ASPENinq_semantics(n,&olds,&oldtl);

          ASPENset_semantics(n,NULL,NULL);

          if ((olds != NULL) && (olds->SEMCOM_last != NULL) &&
              (oldtl != NULL))
            olds->SEMCOM_last->SEMCOM_next = oldtl->SEMCOM_next;

          if ((oldtl != NULL) && (oldtl->SEMCOM_next != NULL) &&
              (olds != NULL))
            oldtl->SEMCOM_next->SEMCOM_last = olds->SEMCOM_last;

          SEMCOM_remove(olds,oldtl);

          if (SEMCOM__auto_recomp) {

            n = enclosing_program(n,TRUE);

            ASPENinq_semantics(n,&hd,&tl);

            SEMCOM_change_complete(hd,tl);
          }

          else

            SEMCOM_change_complete(NULL,NULL);

          break;
      };

    if (_SEMCOM__trace_level & SEMCOM_TRACE_DUMP) _SEMCOM_dump();
};
```

```
/*************************************************************************/
/*                                                                       */
/*      SEMCOM_force_compilation                                         */
/*                                                                       */
/*************************************************************************/


SEMCOM_force_compilation()
{
   ASPEN_NODE n;
   Boolean temp;
   SEMCOM_STMT olds,oldtl,b,hd,tl;

   ITRACE("SEMCOM_force_compilation");

   temp = SEMCOM__auto_recomp;
   SEMCOM__auto_recomp = TRUE;

   switch (SEMCOM__comp_type) {

      case SEMCOM_COMP_INCREMENTAL:

         SEMCOM_record("Attempt to force compilation,");
         SEMCOM_record("   but compilation is set to INCREMENTAL");

         break;

      case SEMCOM_COMP_PROCEDURE:

         SEMCOM_change_procedure(proc_after,proc_old_hd,proc_old_tl,
                                 proc_new_hd,proc_new_tl);

         proc_old_hd = NULL;
         proc_old_tl = NULL;
         proc_new_hd = NULL;
         proc_new_hd = NULL;
         current_procedure = NULL;
         proc_after = NULL;

         break;

      case SEMCOM_COMP_COMPLETE:

         n = enclosing_program(current_node,TRUE);

         ASPENinq_semantics(n,&hd,&tl);

         SEMCOM_change_complete(hd,tl);

         break;
   };
   SEMCOM__auto_recomp = temp;
};
```

Not listed - *_SEMCOM_stmt_free*

Not listed - *_SEMCOM_findprevious*

```
/*************************************************************************/
/*                                                                       */
/*      semcom_new_node — set last focus as current node has changed     */
/*                                                                       */
/*************************************************************************/


static
semcom_new_node(evt,act,node,name,id)
   String evt;
   PLUM_EVENT_ACTION act;
   ASPEN_NODE node;
   String name;
   Integer id;
{
   DTRACE("semcom_new_node");

   if (act != PLUM_EVENT_DO) return;

   previous_node = current_node;
   current_node = node;
};
```

Not listed - *stmt_list*

Not listed - *eval_do_stmt*

Not listed - *new_stmt*

```
/*************************************************************************/
/*                                                                       */
/*      copy_list — copy an existing list of SEMCOM statements into      */
/*                  a new list                                           */
/*                                                                       */
/*************************************************************************/


static
copy_list(old_hd,old_tl,new_hd,new_tl)
   SEMCOM_STMT old_hd,old_tl,*new_hd,*new_tl;
{
   register SEMCOM_STMT s,ls;

   DTRACE("copy_list 0x%x 0x%x",old_hd,old_tl);

   *new_hd = NULL;
   *new_tl = NULL;
   s = NULL;
   ls = NULL;

   while (old_hd != NULL) {

      if (freelist == NULL) s = ALLOC(SEMCOM_STMT);
      else {
         s = freelist;
         freelist = s->SEMCOM_next;
       };

      if (*new_hd == NULL) *new_hd = s;

      s->SEMCOM_next = NULL;
      s->SEMCOM_last = ls;
      s->SEMCOM_type = old_hd->SEMCOM_type;
      s->SEMCOM_index = old_hd->SEMCOM_index;
      s->SEMCOM_node = old_hd->SEMCOM_node;
      s->SEMCOM_value = old_hd->SEMCOM_value;
```

```
        if (ls != NULL) ls->SEMCOM_next = s;

        ls = s;

        old_hd = (old_hd == old_tl ? NULL : old_hd->SEMCOM_next);

      }

    *new_tl = s;
};


/*****************************************************************************/
/*                                                                           */
/*      enclosing_block -- returns the enclosing block node                  */
/*                                                                           */
/*****************************************************************************/


static ASPEN_NODE
enclosing_block(n,print)
    ASPEN_NODE n;
    Boolean print;
{
    ASPEN_NODE x;
    register Integer count;

    DTRACE("enclosing_block 0x%x",n);

    count = 0;
    x = n;

    while ((x != NULL) &&
           (ASPENinq_rule(x) != BLOCK)) {
       x = ASPENinq_parent(x);
       ++count;
       if (x != NULL) n = x;
     };

    if ((x != NULL) && (print))
       SEMCOM_record("Found enclosing block after %d steps",count);

    return n;
};
```

```
/*************************************************************************/
/*                                                                       */
/*      enclosing_program — returns the enclosing program node           */
/*                                                                       */
/*************************************************************************/


static ASPEN_NODE
enclosing_program(n,print)
   ASPEN_NODE n;
   Boolean print;
{
   ASPEN_NODE x;
   register Integer count;

   DTRACE("enclosing_program 0x%x",n);

   count = 0;
   x = n;

   while (x != NULL) {
      x = ASPENinq_parent(x);
      ++count;
      if (x != NULL) n = x;
    };

   if (print)
      SEMCOM_record("Found enclosing program after %d steps",count);

   return n;
};
```

Not listed - *varcmp*

```
/* end of semcomstmt.c */
```

B.6. Abridged Program Listing: *semcomeval.c*

```
/*****************************************************************************/
/*                                                                           */
/*              semcomeval.c                                                 */
/*                                                                           */
/*      Routines for statement list evaluation in symbol compiler            */
/*                                                                           */
/*****************************************************************************/


#include "semcom_local.h"
#include <sem_reader.h>


/*****************************************************************************/
/*                                                                           */
/*      Local storage definitions                                            */
/*                                                                           */
/*****************************************************************************/


        Boolean                 SEMCOM__auto_recomp;
        SEMCOM_COMPILATION_TYPE SEMCOM__comp_type;



/*****************************************************************************/
/*                                                                           */
/*      Miscellaneous definitions                                            */
/*                                                                           */
/*****************************************************************************/


typedef enum {
   EXTEND_STATE_INIT,
   EXTEND_STATE_OLD,
   EXTEND_STATE_NEW,
   EXTEND_STATE_SCAN
} EXTEND_STATE;



/*****************************************************************************/
/*                                                                           */
/*      Forward Definitions                                                  */
/*                                                                           */
/*****************************************************************************/


static                  undo_execution();
static                  do_execution();
static                  head_merge();
static                  tail_merge();
static                  extend();
static                  updateneeds();
static                  insert();
static  Boolean         teststmtmatch();
```

```
/*****************************************************************************/
/*                                                                           */
/*      _SEMCOM_eval_init -- initialize module                               */
/*                                                                           */
/*****************************************************************************/


_SEMCOM_eval_init()
{
   return;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_change_incremental - change a portion of the statement        */
/*                                  list                                     */
/*                                                                           */
/*****************************************************************************/


SEMCOM_change_incremental(after,oldhd,oldtl,newhd,newtl)
   SEMCOM_STMT after;
   SEMCOM_STMT oldhd,oldtl;
   SEMCOM_STMT newhd,newtl;
{
   ITRACE("SEMCOM_change_incremental 0x%x 0x%x 0x%x 0x%x 0x%x",after,
                                                  oldhd,oldtl,newhd,newtl);
   if (!SEMCOM__auto_recomp) {
      SEMCOM_record("Incremental compilation attempted,");
      SEMCOM_record("   but automatic recompilation is OFF");
    }
   else {
      CTRACE("---------- Begin change after 0x%x",after);

      SEMCOM_record
          ("-------------------------------------------------------------------");
      SEMCOM_record("INCREMENTAL COMPILATION");

      head_merge(&after,&oldhd,&oldtl,&newhd,&newtl);
      tail_merge(&after,&oldhd,&oldtl,&newhd,&newtl);
      extend(&after,&oldhd,&oldtl,&newhd,&newtl);

      SEMCOM_remove(oldhd,oldtl);
      insert(after,newhd,newtl);

      CTRACE("---------- End change\n\n");

      SEMCOM_record
          ("-------------------------------------------------------------------");
    };
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_change_procedure - change enclosing procedure                 */
/*                                                                           */
/*****************************************************************************/


SEMCOM_change_procedure(after,oldhd,oldtl,newhd,newtl)
   SEMCOM_STMT after,oldhd,oldtl,newhd,newtl;
{
   ITRACE("SEMCOM_change_procedure 0x%x 0x%x 0x%x 0x%x 0x%x",after,
                                                  oldhd,oldtl,newhd,newtl);

   if (!SEMCOM__auto_recomp) {
      SEMCOM_record("Procedure compilation attempted,");
      SEMCOM_record("    but automatic recompilation is OFF");
    }
   else {
      CTRACE("---------- Begin change");

      SEMCOM_record
          ("-------------------------------------------------------------------");
      SEMCOM_record("PROCEDURE COMPILATION");

      undo_execution(oldhd,oldtl);
      do_execution(after,newhd,newtl);

      CTRACE("---------- End change\n\n");

      SEMCOM_record
          ("-------------------------------------------------------------------");
    };
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_change_complete -- recompile entire program                   */
/*                                                                           */
/*****************************************************************************/

SEMCOM_change_complete(hd,tl)
   SEMCOM_STMT hd,tl;
{
   ITRACE("SEMCOM_change_complete 0x%x 0x%x",hd,tl);

   if (!SEMCOM__auto_recomp) {
      SEMCOM_record("Complete recompilation attempted,");
      SEMCOM_record("    but automatic recompilation is OFF");
    }
   else {
      CTRACE("---------- Begin change");

      SEMCOM_record
          ("-------------------------------------------------------------------");
      SEMCOM_record("COMPLETE RECOMPILATION");

      undo_execution(hd,tl);
      do_execution(NULL,hd,tl);

      CTRACE("---------- End change\n\n");

      SEMCOM_record
          ("-------------------------------------------------------------------");
    };
};
```

Not listed - *SEMCOM_remove*

```
/*****************************************************************************/
/*                                                                           */
/*      undo_execution — unexecute statement list                            */
/*                                                                           */
/*****************************************************************************/


static
undo_execution(hd,tl)
   SEMCOM_STMT hd,tl;
{
   register Integer ct;

   DTRACE("undo_execution 0x%x 0x%x",hd,tl);

   ct = 0;
   while (tl != NULL) {
      _SEMCOM_unexecute(tl);
      ++ct;
      tl = (hd == tl ? NULL : tl->SEMCOM_last);
    };

   CTRACE("Undo %d",ct);

   SEMCOM_record("Undo %d",ct);
};



/*****************************************************************************/
/*                                                                           */
/*      do_execution — execute statement list                                */
/*                                                                           */
/*****************************************************************************/


static
do_execution(after,hd,tl)
   SEMCOM_STMT after;
   SEMCOM_STMT hd,tl;
{
   register Integer ct;

   DTRACE("do_execution 0x%x 0x%x 0x%x",after,hd,tl);

   _SEMCOM_set_current(NULL);
   _SEMCOM_set_current(after);

   ct = 0;
   while (hd != NULL) {
      _SEMCOM_execute(hd);
      ++ct;
      hd = (hd == tl ? NULL : hd->SEMCOM_next);
    };

   CTRACE("Do %d",ct);

   SEMCOM_record("Do %d",ct);
};
```

Not listed - *head_merge*

Not listed - *tail_merge*

Not listed - *extend*

Not listed - *updateneeds*

Not listed - *insert*

Not listed - *teststmtmatch*

```
/* end of semcomeval.c */
```

B.7. Program Listing: *semcomwindow.c*

```
/*****************************************************************************/
/*                                                                           */
/*              semcomwindow.c                                               */
/*                                                                           */
/*      Window routines for incremental symbol compiler                      */
/*                                                                           */
/*****************************************************************************/
/*      James Popple  September/October 1987                                 */

#include "semcom_local.h"


/*****************************************************************************/
/*                                                                           */
/*      Local storage definitions                                            */
/*                                                                           */
/*****************************************************************************/

        ASH_WINDOW              SEMCOM__window;
        Boolean                 SEMCOM__auto_recomp;
        SEMCOM_COMPILATION_TYPE SEMCOM__comp_type;

static  Integer         SEMCOM__vtid = -1;

static  Boolean         eolfg = TRUE;
static  Integer         semcom_font = 0;

static  Integer         num_lines = 0;
static  Integer         num_cols = 0;


/*****************************************************************************/
/*                                                                           */
/*      Forward Definitions                                                  */
/*                                                                           */
/*****************************************************************************/

static                  new_semcom_window();
static                  semcom_control();
static                  setup_semcom_window();
static                  remove_semcom_window();
static                  semcom_define_scroll();
```

```
/*****************************************************************************/
/*                                                                           */
/*      Window Definitions                                                   */
/*                                                                           */
/*****************************************************************************/


static WILLOW_DEFN semcom_window = {
   WILLOW_CLASS_USER,
       { "SEMCOM","pecan.icons",'C' },
       { 300,100, 800,1024, 400,300, 1, 1 },
       { ASH_WINDOW_HIT_PARENT,
         WILLOW_TITLE_TAB_SENSE,
         WILLOW_INSTANCE_SAVED_1 },
       new_semcom_window,
       NULL,
       { { { "COMPILE" },
           WILLOW_LOCATION_TAIL,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_compile) },
         { { "INCREMENTAL" },
           WILLOW_LOCATION_TAIL,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_incremental) },
         { { "PROCEDURE" },
           WILLOW_LOCATION_TAIL,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_procedure) },
         { { "COMPLETE" },
           WILLOW_LOCATION_TAIL,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_complete) },
         { { " AUTO ",NULL,0,0,"MANUAL"},
           WILLOW_LOCATION_TAIL,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_auto) },
         { { "TOP" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_top) },
         { { "BOTTOM" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_bottom) },
         { { "SCROLL UP" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_scroll_up) },
         { { "SCROLL DOWN" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_scroll_down) },
         { { "UP" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_up) },
         { { "DOWN" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_down) },
         { { "CLEAR" },
           WILLOW_LOCATION_BOTTOM,
           WILLOW_BUTTON_NORMAL,
           WILLOW_ACTION_USER(SEMCOM_button_clear) },
         { { "SCROLL" },
           WILLOW_LOCATION_R,
           WILLOW_BUTTON_REGION,
           WILLOW_ACTION_USER(SEMCOM_button_scroll) },
```

```
        { { "Move", "pecan.icons", '1', 0, 0, 1 },
          WILLOW_LOCATION_UL,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_MOVE(DEFAULT) },
        { { "Size", "pecan.icons", '0', 0, 0, 1 },
          WILLOW_LOCATION_UR,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_TYPE_AUX9, NULL, WILLOW_ACTION_DEFAULT },
        { { "Remove" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_ICON(DEFAULT) },
        { { "Push" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_PUSH(DEFAULT) },
        { { "Pop" },
          WILLOW_LOCATION_TITLE,
          WILLOW_BUTTON_NORMAL,
          WILLOW_ACTION_POP(DEFAULT) }
    }
};


/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_window_init -- initialize semcom window                       */
/*                                                                           */
/*****************************************************************************/


SEMCOM_window_init()
{
   ITRACE("SEMCOM_window_init");

   SEMCOM__window = NULL;
   SEMCOM__vtid = NULL;
   SEMCOM__auto_recomp = TRUE;
   SEMCOM__comp_type = SEMCOM_COMP_INCREMENTAL;
   eolfg = TRUE;
   num_lines = 0;
   num_cols = 0;

   WILLOWdefine_window(&semcom_window);
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_scroll -- scroll VT window                                    */
/*                                                                           */
/*****************************************************************************/


SEMCOM_scroll(dl,dc,abs)
   Integer dl,dc;
   Integer abs;
{
   Integer rl,rc;
   Integer cl,cc;
   register Integer b;

   ITRACE("SEMCOM_scroll %d %d %d",dl,dc,abs);

   VT$PUSH(SEMCOM__vtid);
   VT$NO_SCROLL;
   VT$INQ_REGION(&rl,&rc);

   if (dl != 0) {
      rl += dl*(num_lines/4);
    }
   else if (dc != 0) {
      rc += dc*(num_cols/4);
    }
   else {
      VT$INQ_CURRENT(&cl,&cc);
      ITRACE("\tscroll absolute %d %d %d %d %d",rl,rc,cl,cc,abs);
      b = MAX(cl,rl+num_lines);
      b = abs*b/100-num_lines/2;
      if (b < 0) b = 0;
      else if (b > cl) b = cl;
      rl = b;
      rc = 0;
    };

   ITRACE("\tscroll to %d %d",rl,rc);

   VT$REGION(rl,rc);
   VT$POP;

   semcom_define_scroll();
};


/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_clear_screen -- clear VT window                               */
/*                                                                           */
/*****************************************************************************/


SEMCOM_clear_screen()
{
   ITRACE("SEMCOM_clear_screen");

   VT$PUSH(SEMCOM__vtid);
   VT$MOVE(0,0);
   VT$ERASE_SCREEN;
   VT$POP;

   SEMCOM_scroll(0,0,0);
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_record -- put message in transcript for SEMCOM window         */
/*                                                                           */
/*****************************************************************************/


SEMCOM_record(msg,a1,a2,a3)
   String msg;
   Universal a1,a2,a3;
{
   Character buf[256],buf1[256];

   if (SEMCOM__window == NULL) return;

   ITRACE("SEMCOM_record %s",msg);

   VT$PUSH(SEMCOM__vtid);
   VT$SCROLL;
   VT$FONT(semcom_font);

   if (!eolfg) VT$OUT("\n");

   sprintf(buf,msg,a1,a2,a3);
   sprintf(buf1,"%s\n",buf);
   VT$OUT(buf1);

#ifdef VAX
   printf("%s",buf1);
#endif

   VT$POP;

   eolfg = TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      new_semcom_window -- set up semcom window                            */
/*                                                                           */
/*****************************************************************************/


static
new_semcom_window()
{
   register ASH_WINDOW w;

   DTRACE("new_semcom_window");

   w = ASHinq_window();

   SEMCOM__window = w;

   ASHset_control(semcom_control);

   setup_semcom_window();
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      semcom_control -- control message interpreter                        */
/*                                                                           */
/*****************************************************************************/


static
semcom_control(msg,w)
   String msg;
   ASH_WINDOW w;
{
   DTRACE("semcom_control %s 0x%x",msg,w);

   if (STREQL(msg,"PDS$NEXT")) return ASH_CONTROL_OK;

   if (STREQL(msg,"ASH$RESIZE")) {
      setup_semcom_window();
    }
   else if (STREQL(msg,"ASH$INQ_RESIZE")) {
      remove_semcom_window();
    }
   else if (STREQL(msg,"ASH$REMOVE")) {
      remove_semcom_window();
      SEMCOM__window = NULL;
    };

   return ASH_CONTROL_REJECT;
};



/*****************************************************************************/
/*                                                                           */
/*      setup_semcom_window -- set up semcom window                          */
/*                                                                           */
/*****************************************************************************/


static
setup_semcom_window()
{
   DTRACE("setup_semcom_window");

   ASHpush_window();

   ASHselect(SEMCOM__window);

   SEMCOM__vtid = VTopen();
   VT$PUSH(SEMCOM__vtid);
   VT$SCROLL;

   semcom_font = VT$LOADFONT(SEMCOM_FONT);
   VT$FONT(semcom_font);
   VT$INQ_SIZE(&num_lines,&num_cols);

   ASHset_window_name("Compilation monitor");

   VT$POP;

   ASHpop_window();

   semcom_define_scroll();
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      remove_semcom_window -- remove semcom window                         */
/*                                                                           */
/*****************************************************************************/


static
remove_semcom_window()
{
   DTRACE("remove_semcom_window");

   VTclose(SEMCOM__vtid);
};



/*****************************************************************************/
/*                                                                           */
/*      semcom_define_scroll -- define scroll region                         */
/*                                                                           */
/*****************************************************************************/


static
semcom_define_scroll()
{
   Integer rl,rc;
   Integer cl,cc;
   register Integer b;

   DTRACE("semcom_define_scroll");

   VT$PUSH(SEMCOM__vtid);
   VT$INQ_CURRENT(&cl,&cc);
   VT$INQ_REGION(&rl,&rc);
   VT$POP;

   b = MAX(cl,rl+num_lines);
   WILLOWbutton_feedback(SEMCOM__window,"SCROLL",TRUE,
                         WILLOW_SCROLL_REGION(rl*100/b,
                                              (rl+num_lines)*100/b));
};


/* end of semcomwindow.c */
```

B.8. Program Listing: *semcombutton.c*

```
/*****************************************************************************/
/*                                                                           */
/*              semcombutton.c                                               */
/*                                                                           */
/*      Button handling routines for incremental symbol compiler             */
/*                                                                           */
/*****************************************************************************/
/*      James Popple  September/October 1987                                 */

#include "semcom_local.h"


/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_compile -- handle COMPILE button                       */
/*                                                                           */
/*****************************************************************************/

int
SEMCOM_button_compile(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_compile %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_force_compilation();

   return TRUE;
};


/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_incremental -- handle INCREMENTAL button               */
/*                                                                           */
/*****************************************************************************/

int
SEMCOM_button_incremental(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_incremental %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM__comp_type = SEMCOM_COMP_INCREMENTAL;

   SEMCOM_record("Compilation set to INCREMENTAL");

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_procedure -- handle PROCEDURE button                   */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_procedure(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_procedure %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM__comp_type = SEMCOM_COMP_PROCEDURE;

   SEMCOM_record("Compilation set to PROCEDURE");

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_complete -- handle COMPLETE button                     */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_complete(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_complete %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM__comp_type = SEMCOM_COMP_COMPLETE;

   SEMCOM_record("Compilation set to COMPLETE");

   return TRUE;
};
```

```
/*************************************************************************/
/*                                                                       */
/*      SEMCOM_button_auto -- handle AUTO on/off                         */
/*                                                                       */
/*************************************************************************/


int
SEMCOM_button_auto(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_auto %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM__auto_recomp = !SEMCOM__auto_recomp;

   WILLOWbutton_feedback(SEMCOM__window," AUTO ",!SEMCOM__auto_recomp,0);

   SEMCOM_record("Automatic recompilation turned %s",
                 (SEMCOM__auto_recomp ? "ON" : "OFF"));

   return TRUE;
};



/*************************************************************************/
/*                                                                       */
/*      SEMCOM_button_top -- handle TOP button                           */
/*                                                                       */
/*************************************************************************/


int
SEMCOM_button_top(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_top %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(0,0,0);

   return TRUE;
};



/*************************************************************************/
/*                                                                       */
/*      SEMCOM_button_bottom -- handle BOTTOM button                     */
/*                                                                       */
/*************************************************************************/


int
SEMCOM_button_bottom(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_bottom %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(0,0,100);

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_scroll_up -- handle SCROLL UP button                   */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_scroll_up(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_scroll_up %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(-4,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_scroll_down -- handle SCROLL DOWN button               */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_scroll_down(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_scroll_down %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(4,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_up -- handle UP button                                 */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_up(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_up %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(-1,0,0);

   return TRUE;
};
```

```
/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_down -- handle DOWN button                             */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_down(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_down %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(1,0,0);

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_clear -- handle CLEAR button                           */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_clear(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_clear %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_clear_screen();

   return TRUE;
};



/*****************************************************************************/
/*                                                                           */
/*      SEMCOM_button_scroll -- handle scroll bar                            */
/*                                                                           */
/*****************************************************************************/


int
SEMCOM_button_scroll(dir)
   WILLOW_ACTION_MODE dir;
{
   ITRACE("SEMCOM_button_scroll %d",dir);

   if (dir != WILLOW_ACTION_DO) return;

   SEMCOM_scroll(0,0,WILLOWinq_scroll());

   return TRUE;
};


/* end of semcombutton.c */
```

Appendix C
Test Programs

The Pascal programs used for testing in §6.6 are listed in this appendix (§C.1 to §C.4). The program listings have been formatted by PECAN, using the formatting information included in the specification of Pascal (see §5.2).

C.1. Program Listing: *test1.p*

```
PROGRAM matrixproduct (input,output);

    { taken from [Findlay 81], pages 200-201 }

    CONST
        n = 10;

    TYPE
        matrix = ARRAY [ 1 .. n , 1 .. n ] OF integer;

    VAR
        a, b, p : matrix;


PROCEDURE readmatrix  (VAR m : matrix);

    VAR
        i, j : 1 .. n;


BEGIN { Procedure readmatrix }
    FOR i := 1 TO n DO
        FOR j := 1 TO n DO
            READ(m[i,j])
END;


PROCEDURE writematrix  (VAR m : matrix);

    VAR
        i, j : 1 .. n;


BEGIN { Procedure writematrix }
    FOR i := 1 TO n DO
        BEGIN
            WRITE('[');
            FOR j := 1 TO n DO
                WRITE(m[i,j]);
            WRITELN(']')
        END
END;
```

```
PROCEDURE multiplymatrices  (m1, m2 : matrix; VAR product : matrix);

    VAR
        i, j, k : 1 .. n;
        scalarproduct : integer;


BEGIN { Procedure multiplymatrices }
    FOR i := 1 TO n DO
        FOR j := 1 TO n DO
            BEGIN
                scalarproduct := 0;
                FOR k := 1 TO n DO
                    scalarproduct := scalarproduct+m1[i,k]*m2[k,j];
                product[i,j] := scalarproduct
            END
END;


BEGIN { Program matrixproduct }
    readmatrix({ m := } a);
    readmatrix({ m := } b);
    multiplymatrices({ m1 := } a, { m2 := } b, { product := } p);
    writematrix({ m := } p)
END.
```

C.2. Program Listing: *test2.p*

```
PROGRAM tableoftans (output);

    { taken from [Findlay 81], pages 167-168 }

    CONST
        pi = 3.1415926536;

    VAR
        degrees : 0 .. 360;
        line : 0 .. 36;


FUNCTION tan (x : real): real;

    { no declarations }

BEGIN { Function tan }
    tan := sin(x)/cos(x)
END;


BEGIN { Program tableoftans }
    WRITELN('Angle':5 , 'Tangent':15 );
    WRITELN(**);
    FOR line := 0 TO 36 DO
        BEGIN
            degrees := 10*line;
            WRITE(degrees:5 );
            IF degrees MOD 180 = 90 THEN
                WRITELN('Infinity':15 )
            ELSE
                WRITELN(tan({ x := } degrees*pi/180):15 )
        END
END.
```

C.3. Program Listing: *test3.p*

```
PROGRAM factorial (input,output);

    { traditional }

    VAR
        x : integer;


FUNCTION factorial (n : integer): integer;

    { no declarations }

BEGIN { Function factorial }
    IF n = 1 THEN
        factorial := 1
    ELSE
        factorial := n*factorial({ n := } n-1)
END;


BEGIN { Program factorial }
    WRITELN('Enter a number:');
    READLN(x);
    WRITELN(x, '! = ', factorial({ n := } x))
END.
```

C.4. Program Listing: *test4.p*

```
PROGRAM recursivegcd (output);

    { taken from [Jensen 78], page 82 }

    VAR
        x, y, n : integer;


FUNCTION gcd (m, n : integer): integer;

    { no declarations }

BEGIN { Function gcd }
    IF n = 0 THEN
        gcd := m
    ELSE
        gcd := gcd({ m := } n, { n := } m MOD n)
END;


PROCEDURE try  (a, b : integer);

    { no declarations }

BEGIN { Procedure try }
    WRITELN(a, b, gcd({ m := } a, { n := } b))
END;


BEGIN { Program recursivegcd }
    try({ a := } 18, { b := } 27);
    try({ a := } 312, { b := } 2142);
    try({ a := } 61, { b := } 53);
    try({ a := } 98, { b := } 868)
END.
```

Appendix D
Earley's Algorithm

D.1. Introduction

Earley's algorithm is a general context-free parsing algorithm. It handles a larger class of grammars in linear time than most restricted algorithms. For unambiguous grammars it is bounded by n^2 (where n is the number of symbols in the input string). In the worst case its time bound is n^3.

Earley's algorithm was first described in his Ph. D. Thesis [Earley 68]. It is also described in [Aho 72] and (with greater pellucidity) in [Earley 70]. This appendix uses the notation from [Earley 70]. An analysis of the efficiency of the algorithm can be found in that article.

D.2. The Recognizer

A parser must be able to recognize whether an input string is a valid sentence of a given grammar. Earley's recognizer scans, from left to right, an input string $X_1 \ldots X_n$ of symbols, and is able to look ahead some fixed number k of symbols.

While scanning the input string, the recognizer constructs sets (S_i) of states. Each of these state sets is initially empty. Each state s in a state set is a quadruple of the form

$$s = \langle p,\ j,\ f,\ \alpha \rangle.$$

where p is an integer which identifies the production from which the recognizer is attempting to derive the current section of the input string (the productions of the grammar are numbered for this purpose),

j is an integer referring to a place within the right hand side of the production p (this indicates how much of the production has been scanned),

f is an integer referring to the position in the input string where the recognizer first began to look for this instance of the production p,

and α is a k-symbol string which is syntactically allowed to follow this instance of the production p.

It is necessary to ensure that there will always be k symbols for the recognizer to see when looking ahead, even when the input string is fully scanned. To achieve this, a terminating symbol $\dashv$ is introduced[1] and $k+1$ terminating symbols are placed at the right end of the input string.

The recognizer starts by inventing a new production (production 0)

ϕ --> R$\dashv$

where ϕ is a new non-terminal symbol and R is the root of the grammar (the non-terminal which produces a sentence).
A state s is put into the state set S_0 so that

$s = \langle 0,\ 0,\ 0,\ \dashv^k \rangle$

where $\dashv^k$ is a string of k terminating symbols.

For clarity, states will be represented as the pth production with a dot[2] marking the position of the pointer j, together with an integer (the value of f) and a k-symbol string (α). So, the state s can be represented as

ϕ --> .R$\dashv$ 0 $\dashv^k$

D.3. The Recognizer's Operations

The recognizer processes the states in the state set S_i in order, using only three operations: *predictor*, *scanner* and *completer*. These operations are applied to a state s in the following ways:

---

[1] $\dashv$ is a metasymbol; it does not occur in the grammar.

[2] Another metasymbol.

Predictor

If there is a non-terminal symbol to the right of the dot in the production, add a new state to S_i for each alternative production of that non-terminal. Each of these new states has

- its dot at the beginning of the production (as none of the symbols of the production has yet been scanned)
- its f assigned to i (the current position in the input string)
- its α assigned to the k symbols that follow the non-terminal (these are determined by reference to the production in s and/or the value of α in s).

Scanner

If there is a terminal symbol to the right of the dot in the production, compare that terminal symbol with the symbol X_{i+1} (the next symbol in the input string). If they match, add to S_{i+1} a copy of s with

- its dot moved to the right to indicate that the terminal symbol has been scanned
- its f unchanged
- its α unchanged.

Completer

If the dot is at the end of a production, compare α with X_{i+1} ... X_{i+k} (the next k symbols of the input string). If they match, go back to the state set where the recognizer first began to look for this instance of the production (ie. S_f). Take all of the states which could have led to the current production (ie. those states with the same non-terminal to the right of the dot as is on the left hand side of the production in s). Copy these states from S_f into S_i, modified so that each of the new states has

- its dot moved to the right to indicate that the non-terminal symbol has been scanned
- its f unchanged
- its α unchanged.

Each of these operations is applied in turn to the states in S_i, then the recognizer processes the states in S_{i+1}. If applying all three operations to S_i leaves S_{i+1} empty then the input string is not a valid sentence of the language. This means that Earley's algorithm shares the property with some (but not all) other parsing algorithms that as soon as a point is reached in the input string such that no possible following symbols could make the input string a valid sentence of the grammar, the recognizer realizes that the input string is not well-formed.

If the recognizer ever produces a state set S_{i+1} consisting only of the state

ϕ --> R⊣. 0 $\dashv^k$

then the input string is a valid sentence of the grammar.

D.4. Application of the Recognizer to an Example Grammar

Consider the grammar G defined in Figure D-1.[3]

E --> T+E
E --> T
T --> F*T
T --> F
F --> (E)
F --> a

Figure D-1: Definition of the Grammar G

The terminal symbols of the grammar G are {a,+,*,(,)}. The non-terminals are {E, T, F}. Let the input string (X_1 ... X_n) be

(a+a)*a

Let k=1, so that the recognizer will only look one symbol ahead when scanning the input string.

As the root of grammar is E, the recognizer puts the following state into S_0

ϕ --> .E⊣ 0 ⊣

before starting the repeated application of the three operations.

---

[3]Example grammar G is taken from the description of Earley's algorithm in [Aho 72].

To the right of the dot is a non-terminal symbol, so the *predictor* is used. The *predictor* adds a new state to S_0 for each alternative production of E, namely

```
E --> .T+E     0     ⊣
E --> .T       0     ⊣
```

The dots are at the beginning of the productions because none of the symbols has been scanned yet. Each α=⊣ since a ⊣ is to be found after E in the original state. The *predictor* is applied to the two new states. This results in the following states being added to S_0

```
T --> .F*T     0     +
T --> .F       0     +
T --> .F*T     0     ⊣
T --> .F       0     ⊣
```

The *predictor* is applied repeatedly to the states in S_0 until all of the newly-created states have been processed, at which stage S_0 will contain the following states

```
φ --> .E⊣      0     ⊣
E --> .T+E     0     ⊣
E --> .T       0     ⊣
T --> .F*T     0     +
T --> .F       0     +
T --> .F*T     0     ⊣
T --> .F       0     ⊣
F --> .(E)     0     *
F --> .a       0     *
F --> .(E)     0     +
F --> .a       0     +
F --> .(E)     0     ⊣
F --> .a       0     ⊣
```

The *scanner* is now applied. As X_1=(, the *scanner* will add to S_1 those states in S_0 with a (to the right of the dot, with each dot moved to the right to indicate that the (has been scanned. S_1 now contains these states

```
F --> (.E)     0     *
F --> (.E)     0     +
F --> (.E)     0     ⊣
```

The *predictor* is applied to all of the states in S_1 as they all have a non-terminal to the right of the dot. Repeated application of the *predictor* leaves S_1 containing the following states

F --> (.E)	0	*
F --> (.E)	0	+
F --> (.E)	0	⊣
E --> .T+E	1	)
E --> .T	1	)
T --> .F*T	1	+
T --> .F	1	+
T --> .F*T	1	)
T --> .F	1	)
F --> .(E)	1	*
F --> .a	1	*
F --> .(E)	1	+
F --> .a	1	+
F --> .(E)	1	)
F --> .a	1	)

The *scanner* can be applied again. X_2=a, so the *scanner* will add to S_2 every state in S_1 with an a to the right of the dot (the dot in the production in each new state is moved to the right). S_2 now contains the states

F --> a.	1	*
F --> a.	1	+
F --> a.	1	)

The *completer* can now be applied for the first time. Each of the states in S_2 has a dot at the end of its production, but only the second state in S_2 has an α which matches the lookahead string (as k=1, the lookahead string is "+" (ie. X_3)). The *completer* goes back to the state set where the recognizer first began to look for this instance of the production (pointed to by f). As f=1, the *completer* goes back to S_1. Now the *completer* adds to S_2 all those states in S_1 that could have led to the second production in S_2, with the dot moved to the right to indicate that the non-terminal (F) has been successfully scanned. So, the *completer* will add the following states to S_2

T --> F.*T	1	+
T --> F.	1	+
T --> F.*T	1	)
T --> F.	1	)

The *completer* is applied again to the second of these new states as its α matches the lookahead string. This step adds to S_2 the following states from S_1

E --> T.+E	1	)
E --> T.	1	)

The *completer* cannot be applied again to S_2, so the recognizer continues with the application of the *scanner* to the states in S_2.

The recognizer will continue in the manner described above until it produces a state set[4] which contains only the state

ϕ --> E⊣. 0 ⊣

As E is the root of the grammar G, the recognizer has reached the stage where the input string

(a+a)*a

has been recognized as a valid sentence of the grammar. The complete series of state sets for this example appears in Figure D-2.

D.5. Constructing a Parser from the Recognizer

To construct a parser, the recognizer must be modified so that it builds a derivation tree during the recognition process. This is achieved by building links between states when the *completer* operation is used. (For the purposes of building the derivation tree, the values of α can be ignored; lookahead is only required for the recognizer.)

Whenever the *completer* adds a state to a state set, the parser builds a pointer from the non-terminal (before the dot in the new state) to the state which triggered the *completer* operation (which has a production for that non-terminal). If the non-terminal is ambiguous then more than one state will cause the *completer* operation to add the same new state. In that case, there will be a *set* of pointers from the non-terminal in the new state (one for each *completer* operation which added that new state).

When the whole input string has been scanned, the derivation tree for the sentence will be attached to the final state

ϕ --> R⊣. 0

If the sentence that is scanned is ambiguous then all possible derivation trees will be attached to the final state.

---

[4]The final state set is S_{n+1} (in this example S_8).

Input string = (a+a)*a

k=1

S_0
X_1=(

ϕ --> .E⊣	0	⊣
E --> .T+E	0	⊣
E --> .T	0	⊣
T --> .F*T	0	+
T --> .F	0	+
T --> .F*T	0	⊣
T --> .F	0	⊣
F --> .(E)	0	*
F --> .a	0	*
F --> .(E)	0	+
F --> .a	0	+
F --> .(E)	0	⊣
F --> .a	0	⊣

S_1
X_2=a

F --> (.E)	0	*
F --> (.E)	0	+
F --> (.E)	0	⊣
E --> .T+E	1	)
E --> .T	1	)
T --> .F*T	1	+
T --> .F	1	+
T --> .F*T	1	)
T --> .F	1	)
F --> .(E)	1	*
F --> .a	1	*
F --> .(E)	1	+
F --> .a	1	+
F --> .(E)	1	)
F --> .a	1	)

S_2
X_3=+

F --> a.	1	*
F --> a.	1	+
F --> a.	1	)
T --> F.*T	1	+
T --> F.	1	+
T --> F.*T	1	)
T --> F.	1	)
E --> T.+E	1	)
E --> T.	1	)

S_3
X_4=a

E --> T+.E	1	)
E --> .T+E	3	)
E --> .T	3	)
T --> .F*T	3	+
T --> .F	3	+
T --> .F*T	3	)
T --> .F	3	)
F --> .(E)	3	*
F --> .a	3	*
F --> .(E)	3	+
F --> .a	3	+
F --> .(E)	3	)
F --> .a	3	)

S_4
X_5=)

F --> a.	3	*
F --> a.	3	+
F --> a.	3	)
T --> F.*T	3	+
T --> F.	3	+
T --> F.*T	3	)
T --> F.	3	)
E --> T.+E	3	)
E --> T.	3	)
E --> T+E.	1	)
F --> (E.)	0	*
F --> (E.)	0	+
F --> (E.)	0	⊣

S_5
X_6=*

F --> (E).	0	*
F --> (E).	0	+
F --> (E).	0	⊣
T --> F.*T	0	+
T --> F.	0	+
T --> F.*T	0	⊣
T --> F.	0	⊣

Figure D-2: State Sets for the Example Input String

(continued next page)

S_6 X_7=a			S_7 X_8=⊣			S_8		
T --> F*.T	0	+	F --> a.	6	*	ϕ --> E⊣.	0	⊣
T --> F*.T	0	⊣	F --> a.	6	+			
T --> .F*T	6	+	F --> a.	6	⊣			
T --> .F	6	+	T --> F.*T	6	+			
T --> .F*T	6	⊣	T --> F.	6	+			
T --> .F	6	⊣	T --> F.*T	6	⊣			
F --> .(E)	6	*	T --> F.	6	⊣			
F --> .a	6	*	T --> F*T.	0	+			
F --> .(E)	6	+	T --> F*T.	0	⊣			
F --> .a	6	+	E --> T.+E	0	⊣			
F --> .(E)	6	⊣	E --> T.	0	⊣			
F --> .a	6	⊣	ϕ --> E.⊣	0	⊣			

Figure D-2 continued

In the example given in §D.4 the *completer* operation is first applied to the state

F --> a. 1

in S_2, and the following states are added to S_2

T --> F.*T 1
T --> F. 1

The parser builds two pointers (one from the F in each of the new states) to the state

F --> a. 1

A diagram showing the way in which the parser links the states for the whole input string appears in Figure D-3. Although there are several states which are pointed to by more than one other state, there is only one derivation tree attached to the final state (ϕ --> E⊣). (If the grammar G had been defined ambiguously in that it provided more than one way to parse the input string then the parser would have attached to the final state one derivation tree for each alternative derivation of the input string.) Following the pointers from the final state, the parse tree for the whole sentence can be constructed. The parse tree for the example input string is shown in Figure D-4.

Input string = (a+a)*a

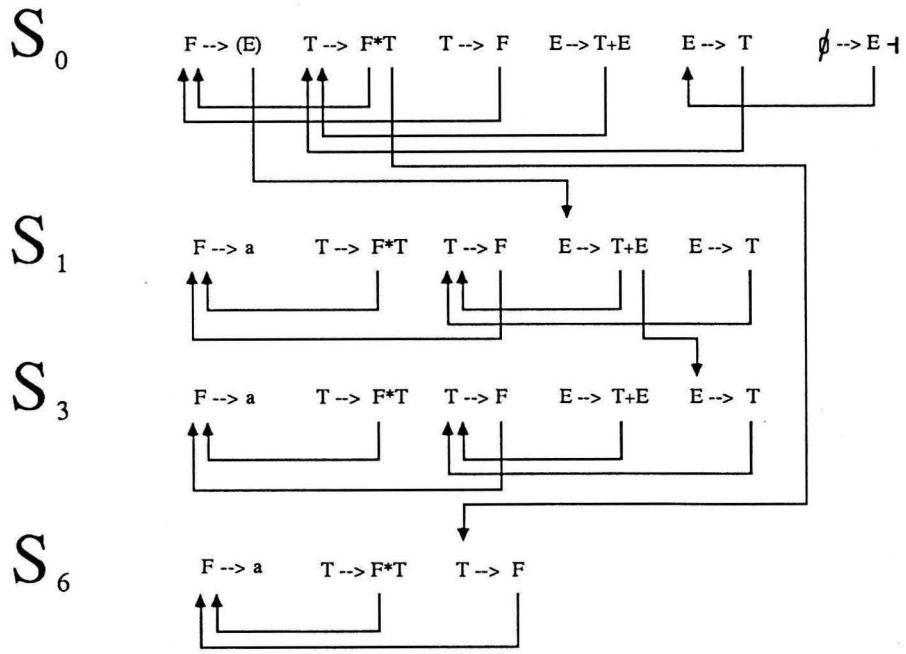

Figure D-3: Linked States for the Example Input String

Input string = (a+a)*a

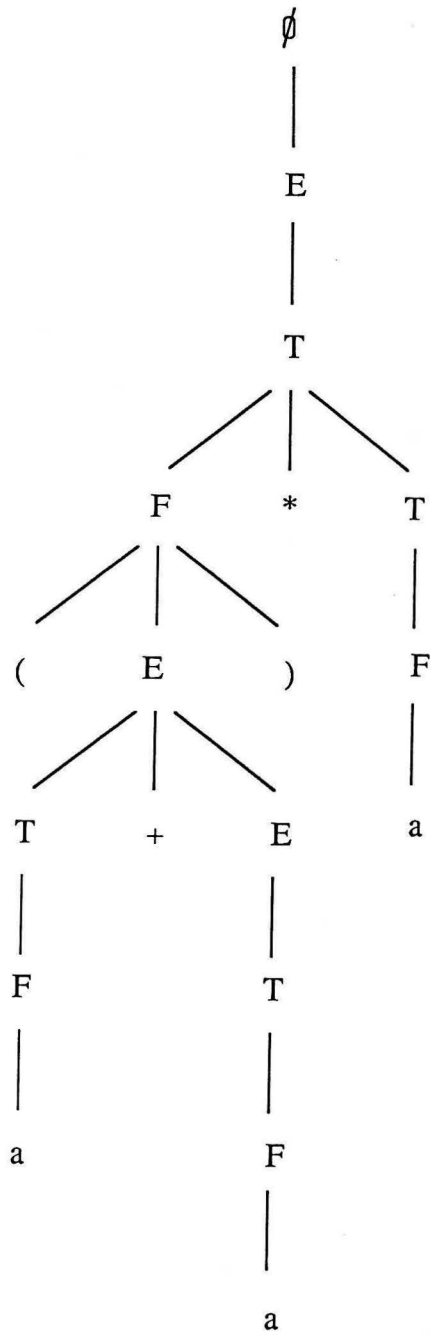

Figure D-4: Parse Tree for the Example Input String

References

[Aho 72] Alfred V. Aho and Jeffrey D. Ullman.
The Theory of Parsing, Translation, and Compiling, Volume I: Parsing.
Prentice-Hall, Englewood Cliffs, New Jersey, 1972.
ISBN 0-13-914556-7.

[Aho 73] Alfred V. Aho and Jeffrey D. Ullman.
The Theory of Parsing, Translation, and Compiling, Volume II: Compiling.
Prentice-Hall, Englewood Cliffs, New Jersey, 1973.
ISBN 0-13-914564-8.

[Aho 86] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman.
Compilers - Principles, Techniques, and Tools.
Addison-Wesley, Reading, Massachusetts, 1986.
ISBN 0-201-10088-6.

[Atkinson 78] L.V. Atkinson and J.J. McGregor.
CONA - A Conversational Algol System.
Software - Practice and Experience 8(6):699-708, November-December 1978.

Describes an implementation of conversational Algol. CONA converts Algol programs into an intermediate form which can be interpreted efficiently. When a change is made, CONA recompiles the entire program (the intermediate representation and the new text) into the intermediate form.

[Atkinson 81a] L.V. Atkinson and S.D. North.
COPAS - A Conversational Pascal System.
Software - Practice and Experience 11(8):819-829, August 1981.

Describes an implementation of conversational Pascal, analogous to CONA [Atkinson 78].

[Atkinson 81b] L.V. Atkinson, J.J. McGregor and S.D. North.
Context sensitive editing as an approach to incremental compilation.
The Computer Journal 24(3):222-229, August 1981.

Describes the implementation of an incremental system for developing programs in a subset of Algol-60. A syntax-directed editor is used, and machine code is produced.

[Bahlke 86] Rolf Bahlke and Gregor Snelting.
The PSG System: From Formal Language Definitions To Interactive Programming Environments.
ACM TOPLAS 8(4):547-576, October 1986.

[Barlow 86a] J. Barlow, S. Leung, M. Nearhos and D. Purdue.
PECAN Programming Environment User Guide.
Paper, Department of Computer Science, Australian National University, Canberra, 19 November 1986.

[Barlow 86b] John D. Barlow.
Generation of a Programming Environment for Concurrent Languages using the PECAN Programming Environment Generator.
Honours Thesis, Department of Computer Science, Australian National University, Canberra, November 1986.

[Bazik 85] John Bazik, Joseph N. Pato, Steven P. Reiss and Marc H. Brown.
The Brown Workstation Environment Programmer's Manual, Version 1.0.
Manual, Department of Computer Science, Brown University, Providence, Rhode Island, January 1985.

[Braden 68] Helen V. Braden and William A. Wulf.
The Implementation of a BASIC System in a Multiprogramming Environment.
Communications of the ACM 11(10):688-692, October 1968.
Gives details of a simple incremental compiler for BASIC programs. Most statements are compiled into machine code, then statement-to-statement execution is handled interpretively.

[Brown 79] P.J. Brown.
Writing Interactive Compilers and Interpreters.
John Wiley and Sons, Chichester, 1979.
(Reprinted with corrections 1980.)
ISBN 0-471-27609-X.
A practical guide to implementing interactive languages. Unfortunately, its discussion of incremental compilation is limited to languages with few context-dependent features.

[Chandhok 85] Ravinder Chandhok, David Garlan, Dennis Goldenson, Philip Miller and Mark Tucker.
Programming environments based on structure editing: The GNOME approach.
in Anthony S. Wojcik (ed.)
1985 *National Computer Conference, AFIPS Conference Proceedings, Volume* 54.
AFIPS Press, Reston, Virginia, 1985.
ISBN 0-88283-046-5.
pp. 359-369
See also [Garlan 84].

[Crowe 82] M.K. Crowe.
An Incremental Compiler.
ACM SIGPLAN Notices 17(10):13-22, October 1982.
Gives details of an experimental incremental system. This paper would appear to add nothing to previously published work.

[Crowe 85] Malcolm Crowe, Clark Nicol, Michael Hughes and David Mackay.
On Converting a Compiler into an Incremental Compiler.
ACM SIGPLAN Notices 20(10):14-22, October 1985.

Describes a strategy for incremental parsing based on dynamic maintainance (sic) of a program's syntax tree. Unfortunately, the authors give a very simple (and simplistic) description of their method, which has been used to develop an incremental parser for Ada.

[Delisle 84] Norman M. Delisle, David E. Menicosy and Mayer D. Schwartz.
Viewing a Programming Environment as a Single Tool.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, *ACM SIGPLAN Notices* 19(5):49-56, May 1984.

[Demers 81] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Application to Syntax-directed Editors.
in Conference Record of the Eighth Annual ACM Symposium on POPL.
ACM, New York, New York, 1981.
ISBN 0-89791-029-X.
pp. 105-116

Discusses the uses of attribute grammars for specifying syntax-directed editors. An algorithm is provided for evaluating attributes incrementally.

[Earley 68] Jay Earley.
An Efficient Context-Free Parsing Algorithm.
Ph. D. Thesis, Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1968.

[Earley 70] Jay Earley.
An Efficient Context-Free Parsing Algorithm.
Communications of the ACM 13(2):94-102, February 1970.

Based on [Earley 68].

[Earley 72] Jay Earley and Paul Caizergues.
A Method for Incrementally Compiling Languages with Nested Statement Structure.
Communications of the ACM 15(12):1040-1044, December 1972.

[Findlay 81] William Findlay and David A. Watt.
Pascal: An Introduction to Methodical Programming (second edition).
Pitman, London, 1981.
ISBN 0 273 01714 4.

[Ford 84] Ray Ford and Duangkaew Sawamiphakdi.
A Greedy Concurrent Approach to Incremental Code Generation.
in Conference Record of the Twelfth Annual ACM Symposium on POPL, 1985.
ACM, New York, New York, 1984.
ISBN 0-89791-147-4.
pp. 165-178

Describes the PSEP system (Parallel Syntax-directed Editing for Pascal) which uses two processes (one for the editor and one for the code generator) to create an incremental system.

[Fritzson 83a] Peter Fritzson.
Symbolic Debugging Through Incremental Compilation in an Integrated Environment.
The Journal of Systems and Software 3(4):285-294, December 1983.

[Fritzson 83b] Peter Fritzson.
A Systematic Approach to Advanced Debugging through Incremental Compilation, Preliminary Draft.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, ACM SIGPLAN Notices 18(8):130-139, August 1983.

[Garlan 84] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5):65-72, May 1984.
Describes a suite of programming environments built around syntax-directed editors. A family tree language, a language for moving a simulated robot around a grid, Pascal and Fortran are supported.

[Ghezzi 79] Carlo Ghezzi and Dino Mandrioli.
Incremental Parsing.
ACM TOPLAS 1(1):58-70, July 1979.
Gives an algorithm for an incremental LR parser, and a few general suggestions as to how it could be implemented. Unfortunately, the language and the notation used in this article combine to make it virtually incomprehensible.

[Ghezzi 80] Carlo Ghezzi and Dino Mandrioli.
Augmenting Parsers to Support Incrementality.
Journal of the ACM 27(3):564-579, July 1980.
Describes a method of inserting incremental parsing into a shift-reduce parsing algorithm. This paper shares the abstruseness of [Ghezzi 79].

[Goldberg 83] Adele Goldberg and David Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, Reading, Massachusetts, 1983.
ISBN 0-201-11371-6.

[Goldberg 84] Adele Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, Reading, Massachusetts, 1984.
ISBN 0-201-11372-4.

[Heyman 85] Jerrold Heyman and William M. Lively.
Syntax-Directed Editing Revisited.
ACM SIGSOFT Software Engineering Notes 10(3):24-27, July 1985.
Gives details of a fairly simple syntax-directed editor. It is hard to see exactly what this paper, and the system it describes, have added to syntax-directed editor technology.

[Jensen 78] Kathleen Jensen and Niklaus Wirth.
Pascal User Manual and Report.
Springer-Verlag, New York, 1978.
ISBN 0-387-90144-2.

[Johnson 87] C.W. Johnson and B.P. Molinari.
Generated Symbol Analysis for Languages with Explicit Scope Control.
Paper, Department of Computer Science, Australian National University, Canberra, 23 September 1987.

[Kahrs 79] Mark Kahrs.
Implementation of an Interactive Programming System.
Proceedings of the ACM SIGPLAN Symposium on Compiler Construction, ACM SIGPLAN Notices 14(8):76-82, August 1979.
Describes a system where those lines of a program which are altered are simply tagged for recompilation at execution time.

[Kaiser 85] Gail E. Kaiser and Elaine Kant.
Incremental Parsing without a Parser.
The Journal of Systems and Software 5(2):121-144, May 1985.
Describes a system where a syntax-directed editor's representation of a program can be incrementally modified not by re-parsing, but by tree transformation.

[Kastens 82] Uwe Kastens, Brigitte Hutt and Erich Zimmermann.
GAG: A Practical Compiler Generator.
G. Goos and J. Hartmanis (eds)
Lecture Notes in Computer Science, Number 141.
Springer-Verlag, Berlin, 1982.
ISBN 3-540-11591-9.

[Katzan 69] Harry Katzan, Jr.
Batch, conversational, and incremental compilers.
in 1969 *Spring Joint Computer Conference, AFIPS Conference Proceedings, Volume* 34.
AFIPS Press, Montvale, New Jersey, 1969.
pp. 47-56.

[Kernighan 78] Brian W. Kernighan and Dennis M. Ritchie.
The C Programming Language.
Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
ISBN 0-13-110163-3.

[Ledgard 81] Henry Ledgard and Micheal Marcotty.
The Programming Language Landscape.
Science Research Associates, Chicago, 1981.
ISBN 0-574-21340-6.

[Leung 86] Sek Kit Leung.
Generation of Modula-2 Programming Environment Using the PECAN System.
Honours Thesis, Department of Computer Science, Australian National University, Canberra, November 1986.

[Medina-Mora 81]
Raul Medina-Mora and Peter H. Feiler.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September 1981.

[Molinari 85] B.P. Molinari.
PLUM - A Package for ADT Implementation.
Technical Report TR-CS-85-03, Department of Computer Science, Australian National University, Canberra, December 1985.

[Molinari 86] B. Molinari.
ASPEN: a module for AST support.
Paper, Department of Computer Science, Australian National University, Canberra, September 1986.

[Molinari 87a] B.P. Molinari.
Notes on PECAN's specification language.
Notes, Department of Computer Science, Australian National University, Canberra, 30 January 1987.

[Molinari 87b] Brian P. Molinari and Christopher W. Johnson.
Generation of Symbol Processing Modules.
Technical Report TR-CS-87-02, Department of Computer Science, Australian National University, Canberra, June 1987.

[Morris 81] Joseph M. Morris and Mayer D. Schwartz.
The Design of a Language-Directed Editor for Block-Structured Languages.
Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, ACM SIGPLAN Notices 16(6):28-33, June 1981.
A clear discussion of some of the matters to be considered when designing a syntax-directed editor; how such an editor must balance the need to maintain the syntactic structure of the text with the need for editing flexibility. Briefly discusses the connection between syntax-directed editors and incremental parsers.

[Nearhos 86] Mandy F. Nearhos.
A Program Profiler in the PECAN Programming Environment.
Grad. Dip. Thesis, Department of Computer Science, Australian National University, Canberra, November 1986.

[Nordström 84] Bengt Nordström and Åke Wikström.
The Design of an Interactive Program Development System for Pascal.
Software - Practice and Experience 14(2):177-190, February 1984.

[Ophel 87] John Ophel.
A Survey of Programming Environments and Some Comments on Their Design.
Paper, Department of Computer Science, Australian National University, Canberra, 27 February 1987.

[Parker 85] Jeff Parker.
Towards More Intelligent Programming Environments.
ACM SIGSOFT Software Engineering Notes 10(3):28-32, July 1985.

A brief discussion of the possible future development of programming environments. The author suggests that programming environments should not be used only for developing programs in existing languages, but that new interactive programming languages should be developed to make full use of these environments.

[Peccoud 69] M. Peccoud, M. Griffiths and M. Peltier.
Incremental Interactive Compilation.
in A.J.H. Morrell (ed.)
Information Processing 68, *Proceedings of IFIP Congress* 1968, *Volume* 1 - *Mathematics, Software.*
North-Holland, Amsterdam, 1969.
pp. 384-387.

[Pollock 84] Lori L. Pollock and Mary Lou Soffa.
Incremental Compilation of Locally Optimized Code.
in Conference Record of the Twelfth Annual ACM Symposium on POPL, 1985.
ACM, New York, New York, 1984.
ISBN 0-89791-147-4.
pp. 152-164.

[Purdue 86] David A. Purdue.
A Graphical View of Data Structure Values for the PECAN Program Development System.
Honours Thesis, Department of Computer Science, Australian National University, Canberra, November 1986.

[Reiss 83] Steven P. Reiss.
Generation of Compiler Symbol Processing Mechanisms from Specifications.
ACM TOPLAS 5(2):127-163, April 1983.

[Reiss 84a] Steven P. Reiss.
An Approach to Incremental Compilation.
Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19(6):144-156, June 1984.

[Reiss 84b] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5):30-41, May 1984.

[Reps 83] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM TOPLAS 5(3):449-477, July 1983.

Discusses techniques for updating programs represented as attribute trees by syntax-directed editors.

[Reps 84] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices 19(5):42-48, May 1984.

[Reps 85] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
Reference Manual, Department of Computer Science, Cornell University, Ithaca, New York, August 1985.

[Rishel 70] Wesley J. Rishel.
Incremental Compilers.
Datamation 16(1):129-136, January 1970.
Describes incremental compilation techniques which require insertion of extra instructions between statements.

[Schwartz 84] Mayer D. Schwartz, Norman M. Delisle and Vimal S. Begwani.
Incremental Compilation in Magpie.
Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction, ACM SIGPLAN Notices 19(6):122-131, June 1984.

[Swinehart 86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach and Robert B. Hagmann.
A Structural View of the Cedar Programming Environment.
ACM TOPLAS 8(4):419-490, October 1986.

[Teitelbaum 81] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9):563-573, September 1981.

[Teitelman 81] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
Computer 14(4):25-33, April 1981.

[Teitelman 84] Warren Teitelman.
A Tour Through Cedar.
IEEE Software 1(2):44-73, April 1984.

[Tichy 84] Walter F. Tichy.
Smart Recompilation.
in Conference Record of the Twelfth Annual ACM Symposium on POPL, 1985.
ACM, New York, New York, 1984.
ISBN 0-89791-147-4.
pp. 236-244.
An extended abstract of [Tichy 86].

[Tichy 86] Walter F. Tichy.
Smart Recompilation.
ACM TOPLAS 8(3):273-291, July 1986.
Examines incremental compilation issues in a modular system. Describes methods of reducing the set of modules that must be compiled after a change to one module.

[Vegdahl 85] Steven R. Vegdahl.
The Design of an Interactive Compiler for Optimizing Microprograms.
Proceedings of the Eighteenth Annual Workshop on Microprogramming, ACM SIGMICRO Newsletter 16(4):129-135, December 1985.

[Waite 84] William M. Waite and Gerhard Goos.
Compiler Construction.
Springer-Verlag, New York, New York, 1984.
ISBN 0-387-90821-8.

[Wasserman 81] Anthony I. Wasserman.
Tutorial: Software Development Environments.
IEEE Computer Society Press, Los Alamitos, 1981.
A collection of papers on programming environments. Includes a remarkable dedication: *To everyone who is still in favor (sic) of peace, equal justice and opportunity for all, clean air and water, and the separation of church and state.*

[Weinberg 71] Gerald M. Weinberg.
The Psychology of Computer Programming.
Van Nostrand Reinhold Company, New York, New York, 1971.
ISBN 0-442-29264-3.

[Wilander 80] Jerker Wilander.
An Interactive Programming System for Pascal.
BIT 20(2):163-174, 1980.
Describes an incremental programming environment for Pascal, called (unforgivably) *Pathcal*, in which all of the system facilities are Pascal procedures or variables, allowing the programmer to view the entire system as a single paradigm.

[Wirth 83] Niklaus Wirth.
Programming in Modula-2 (second edition).
Springer-Verlag, Berlin, 1983.
ISBN 3-540-12206-0.

Abbreviations used in References

ACM
Association for Computing Machinery
AFIPS
American Federation of Information Processing Societies
BIT
Nordisk Tidskrift for Informations-Behandling
IEEE
Institute for Electrical and Electronic Engineers
IFIP
International Federation for Information Processing
ISBN
International Standard Book Number
POPL
Principles of Programming Languages
SIGMICRO
Special Interest Group on Microprogramming
SIGOA
Special Interest Group on Office Automation
SIGPLAN
Special Interest Group on Programming Languages
SIGSOFT
Special Interest Group on Software Engineering
TOPLAS
Transactions on Programming Languages and Systems

Index